9

Competitive Analysis of Omni-Do in Partitionable Networks

THE efficiency of an algorithm solving the *Omni-Do* problem can only be partially understood through its worst case work analysis, such as we did for algorithm AX in the previous chapter. This is because the worst case upper and lower bounds might depend on unusual or extreme patterns of regroupings. In such cases, worst case work may not be the best way to compare the efficiency of algorithms. Hence, in this chapter, in order to understand better the practical implications of performing work in partitionable settings, we treat the *Omni-Do* problem as an on-line problem and we pursue *competitive analysis*, that is we compare the efficiency of a given algorithm to the efficiency of an "off-line" algorithm that has full knowledge of future changes in the communication medium. We consider asynchronous processors under arbitrary patterns of regroupings (including, but not limited to, fragmentation and merges). A processor crash is modeled as the creation of a singleton group (containing the crashed processor) that remains disconnected for the entire computation; the processors in such groups are charged for completing all remaining tasks, in other words, the analysis assumes the worst case situation where a crashed processor becomes disconnected, but manages to complete all tasks before the crash.

In this chapter we view algorithms as a rule that, given a group of processors and a set of tasks known by this group to be completed, determines a task for the group to complete next. We assume that task executions are atomic with respect to regroupings (a task considered for execution by a group is either executed or not prior a subsequent regrouping). Processors in the same group can share their knowledge of completed tasks and, while they remain connected, avoid doing redundant work. The challenge is to avoid redundant work "globally", in the sense that processors should be performing tasks with anticipation of future changes in the network topology. An optimal algorithm, with full knowledge of the future regroupings, can schedule the execution of the tasks in each group in such a way that the overall task-oriented work is the smallest possible, given the particular sequence of regroupings.

As an example, consider the scenario with 3 processors that, starting from isolation, are permitted to proceed synchronously until each has completed $n/2$ tasks; at this point an adversary chooses a pair of processors to merge into a group. It is easy to show that if N_1, N_2, and N_3 are subsets of $[n]$ of size $n/2$, then there is a pair (N_i, N_j) (where $i \neq j$) so that $|N_i \cap N_j| \geq n/6$: in particular, for *any* scheduling algorithm, there is a pair of processors which, if merged at this point, will have $n/6$ duplicated tasks; this pair alone must then expend $n + n/6$ task-oriented work to complete all n tasks. The optimal off-line algorithm that schedules tasks with full knowledge of future merges, of course, accrues only n task-oriented work for the merged pair, as it can arrange for zero overlap. Furthermore, if the adversary partitions the two merged processors immediately after the merge (after allowing the processors to exchanged information about task executions), then the task-oriented work performed by the merged and then partitioned pair is $n + n/3$; the task-oriented work performed by the optimal algorithm remains unchanged, since it terminates at the merge.

To focus on scheduling issues, we assume that processors in a single group work as a single virtual unit; indeed, we treat them as a single asynchronous processor. To this respect, we assume that communication within groups is instantaneous and reliable. We note that the above assumptions can be approximated by group communication services (such as the one considered in Section 8.2) if the reconfiguration time during which no tasks are performed is disregarded. However, in large scale wide-area networks the time performance (which we do not consider here) of *Omni-Do* algorithms can be negatively affected, as GCSs can be inefficient in such networks.

Chapter structure.

In Section 9.1 we present the model of adversity considered in this chapter, we define the notion of competitiveness and we present terminology borrowed from set theory and graph theory that we use in the rest of the chapter. In Section 9.2 we formulate a simple randomized algorithm, called algorithm RS, and we analyze its competitiveness. A result for deterministic algorithms is also given. In Section 9.3 we present lower bounds on the competitiveness of (deterministic and randomized) algorithms for *Omni-Do*, and we claim the optimality of algorithm RS. We discuss open problems in Section 9.4.

9.1 Model of Adversity, Competitiveness and Definitions

In this section we present Adversary $\mathcal{A}_{GR}$, the adversary assumed in this chapter, we formalize the notion of competitiveness, and we recall graph and set theoretic terminology used in the remainder sections.

9.1.1 Adversary $\mathcal{A}_{GR}$

We denote by $\mathcal{A}_{GR}$ an oblivious (off-line) adversary that can cause arbitrary regroupings. Consider an algorithm A that solves the *Omni-Do* problem under adversary $\mathcal{A}_{GR}$. The adversary determines, prior to the start of an execution of A, both the *sequence of regroupings* and the *number of tasks completed* by each group before it is involved in another regrouping. Taken together, this information determines, what we call, a *computation template*: this is a directed acyclic graph (DAG), each vertex of which corresponds to a group of processors that existed during the the computation; a directed edge is placed from group g_1 to group g_2 if g_2 is created by a regrouping involving g_1. We label each vertex of the DAG with the group of processors associated with that vertex and the total number of tasks that the adversary allows the group of processors to perform before the next reconfiguration occurs.

Specifically, if n is the number of *Omni-Do* tasks and p the number of participating processors, then such a computation template is a labeled and weighted DAG, which we call a (p, n)-DAG. More formally,

Definition 9.1. *A (p, n)-DAG is a DAG $C = (V, E)$ augmented with a weight function $h : V \to [n] \cup \{0\}$ and a labeling $\gamma : V \to 2^{[p]} \setminus \{\emptyset\}$ so that:*

1. *For any maximal path $(v_1, \ldots, v_k)$ in C, $\sum h(v_i) \geq n$. (This guarantees that any algorithm terminates during the computation described by the DAG.)*
2. *γ possesses the following "initial conditions":*

$$[p] = \dot{\bigcup_{v:\ in(v)=0}} \gamma(v).$$

3. *γ respects the following "conservation law":*
   there is a function $\phi : E \to 2^{[p]} \setminus \{\emptyset\}$ so that for each $v \in V$ with $indegree(v) > 0$,

$$\gamma(v) = \dot{\bigcup_{(u,v)\in E}} \phi\big((u, v)\big),$$

   and for each $v \in V$ with $out(v) > 0$,

$$\gamma(v) = \dot{\bigcup_{(v,u)\in E}} \phi\big((v, u)\big).$$

In the above definition, $\dot{\cup}$ denotes disjoint union, and $in(v)$ and $out(v)$ denote the in-degree and out-degree of v, respectively. Finally, for two vertices $u, v \in V$, we write $u \leq v$ if there is a directed path from u to v; we then write $u < v$ if $u \leq v$ and u and v are distinct.

Adversary $\mathcal{A}_{GR}$ is constrained to establish only the computation templates as defined above.

Example 9.2. Consider the $(12, n)$-DAG shown in Figure 9.1, where we let the following groups be represented: $g_1 = \{p_1\}$, $g_2 = \{p_2, p_3, p_4\}$, $g_3 = \{p_5, p_6\}$, $g_4 = \{p_7\}$, $g_5 = \{p_8, p_9, p_{10}, p_{11}, p_{12}\}$, $g_6 = \{p_1, p_2, p_3, p_4, p_6\}$, $g_7 = \{p_8, p_{10}\}$, $g_8 = \{p_9, p_{11}, p_{12}\}$, $g_9 = \{p_1, p_2, p_3, p_4, p_6, p_8, p_{10}\}$, $g_{10} = \{p_5, p_{11}\}$, and $g_{11} = \{p_9, p_{12}\}$.

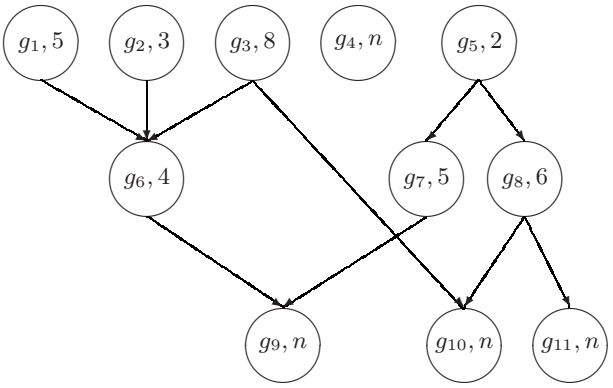

Fig. 9.1. An example of a $(12, n)$-DAG.

This computation template models all (asynchronous) computations with the following behavior.

(*i*) The processors in groups g_1 and g_2 and processor p_6 of group g_3 are regrouped during some reconfiguration to form group g_6. Processor p_5 of group g_3 becomes a member of group g_{10} during the same reconfiguration (see below). Prior to this reconfiguration, processor p_1 (the singleton group g_1) has performed exactly 5 tasks, the processors in g_2 have cooperatively performed exactly 3 tasks and the processors in g_3 have cooperatively performed exactly 8 tasks (assuming that $t > 8$).

(*ii*) Group g_5 is partitioned during some reconfiguration into two new groups, g_7 and g_8. Prior to this reconfiguration, the processors in g_5 have performed exactly 2 tasks.

(*iii*) Groups g_6 and g_7 merge during some reconfiguration and form group g_9. Prior to this merge, the processors in g_6 have performed exactly 4 tasks (counting only the ones performed after the formation of g_6 and assuming that there are at least 4 tasks remaining to be done) and the processors in g_7 have performed exactly 5 tasks.

(*iv*) The processors in group g_8 and processor p_5 of group g_3 are regrouped during some reconfiguration into groups g_{10} and g_{11}. Prior to this reconfiguration, the processors in group g_8 have performed exactly 6 tasks (assuming that there are at least 6 tasks remaining, otherwise they would have performed the remaining tasks).

(v) The processors in g_9, g_{10}, and g_{11} run until completion with no further reconfigurations.

(vi) Processor p_7 (the singleton group g_4) runs in isolation for the entire computation.

Given a (p, n)-DAG representing a computation template C, we say that two vertices (representing groups) are *independent* if there is no direct path connecting one to the other. Then, for the computation template C we define the *computation width of* C, $\mathbf{cw}(C)$, to be the maximum number of independent vertices reachable from any vertex in (p, n)-DAG. We give a formal definition at the conclusion of this section.

Let ξ is the execution of an algorithm solving *Omni-Do* under the computation template C represented by a (p, n)-DAG. We let the adversarial pattern $\xi|_{\mathcal{A}_{GR}}$ be represented by the (p, n)-DAG, or its appropriate subgraph[1]. Following the notation established in Section 2.2.3, we define the weight of $\xi|_{\mathcal{A}_{GR}}$ as the computation width of this graph, that is, $\|\xi|_{\mathcal{A}_{GR}}\| \leq \mathbf{cw}(C)$. (From the definition of the computation width it is easy to observe that given a subgraph H of a DAG G, $\mathbf{cw}(H) \leq \mathbf{cw}(G)$.)

9.1.2 Measuring Competitiveness

Before we formally define the notion of competitiveness, we introduce some terminology.

Let D be a deterministic algorithm for *Omni-Do* and C a computation template. We let $W_{\mathrm{D}}(C)$ denote the task-oriented work expended by algorithm D, where regroupings are determined according to the computation template C. That is, if $\xi \in \mathcal{E}(\mathrm{D}, \mathcal{A}_{GR})$ is an execution of algorithm D under computation template C, then $W_{\mathrm{D}}(C)$ is the task-oriented work of execution ξ. We let OPT denote the optimal (off-line) algorithm, meaning that for each C we have $W_{\mathrm{OPT}}(C) = \min_{\mathrm{D}} W_{\mathrm{D}}(C)$. We now move to define competitiveness.

Definition 9.3. *Let α be a real valued function defined on the set of all (p, n)-DAGs (for all p and n). A deterministic algorithm D is* **α-competitive** *if for all computation templates C,*

$$W_D(C) \leq \alpha(C) W_{OPT}(C).$$

In this chapter we treat randomized algorithms as distributions over deterministic algorithms; for a set Z and a family of deterministic algorithms $\{\mathrm{D}_\zeta \mid \zeta \in Z\}$ we let $\mathrm{R} = \mathcal{R}(\{\mathrm{D}_\zeta \mid \zeta \in Z\})$ denote the randomized algorithm where ζ is selected uniformly at random from Z and scheduling is done according to D_ζ. For a real-valued random variable X, we let $\mathbb{E}[X]$ denote its expected value. Then,

[1] The execution might terminate with all tasks performed before all regroupings specified by the computation template take place; this is possible in the case of randomized algorithm where the oblivious adversary does not know *a priori* how the algorithm would behave under the specific sequence of regroupings.

Definition 9.4. *Let α be a real valued function defined on the set of all (p, n)-DAGs (for all p and n). A randomized algorithm R is* **α-competitive** *if for all computation templates C,*

$$\mathbb{E}[W_{\mathrm{D}_\zeta}(C)] \leq \alpha(C) W_{OPT}(C),$$

this expectation being taken over uniform choice of $\zeta \in Z$.

Note that usually α is fixed for all inputs; we shall see in later sections that this would be meaningless in this setting. Presently, we use a function α that depends on a certain parameter of the graph structure of C, namely the computation width $\mathbf{cw}(C)$.

9.1.3 Formalizing Computation Width

We conclude this subsection with definitions and terminology that we use in the remainder of this chapter.

Definition 9.5. *A* partially ordered set *or* poset *is a pair $(P, \leq)$ where P is a set and $\leq$ is a binary relation on P for which (i) for all $x \in P$, $x \leq x$, (ii) if $x \leq y$ and $y \leq x$, then $x = y$, and (iii) if $x \leq y$ and $y \leq z$, then $x \leq z$. For a poset $(P, \leq)$ we overload the symbol P, letting it denote both the set and the poset.*

Definition 9.6. *Let P be a poset. We say that two elements x and y of P are* comparable *if $x \leq y$ or $y \leq x$; otherwise x and y are* incomparable. *A* chain *is a subset of P such that any two elements of this subset are comparable. An* antichain *is a subset of P such that any two distinct elements of this subset are incomparable. The* width *of P, denoted $\mathbf{w}(P)$, is the size of the largest antichain of P.*

Associated with any DAG $C = (V, E)$ is the natural *vertex poset* $(V, \leq)$ where $u \leq v$ if and only if there is a directed path from u to v. Then the *width of C*, denoted $\mathbf{w}(C)$, is the width of the poset $(V, \leq)$.

Definition 9.7. *Given a DAG $C = (V, E)$ and a vertex $v \in V$, we define the* predecessor graph at v, *denoted $P_C(v)$, to be the subgraph of C that is formed by the union of all paths in C terminating at v. Likewise, the* successor graph at v, *denoted $S_C(v)$, is the subgraph of C that is formed by the union of all the paths in C originating at v.*

Using the above definitions and terminology we give a formal definition of the computation width of a given computation template.

Definition 9.8. *The* computation width *of a DAG $C = (V, E)$, denoted $\mathbf{cw}(C)$, is defined as*

$$\mathbf{cw}(C) = \max_{v \in V} \mathbf{w}(S_C(v)).$$

Note that the processors that comprise a group formed during a computation template C may be involved in many different groups at later stages of the computation, but no more than $\mathbf{cw}(C)$ of these groups can be computing in ignorance of each other's progress.

Example 9.9. In the $(12, n)$-DAG of Figure 9.1, the maximum width among all successor graphs is 3: $\mathbf{w}(S((g_5, 2))) = 3$. Therefore, the computation width of this DAG is 3. Note that the width of the DAG is 6 (nodes $(g_1, 5), (g_2, 3), (g_3, 8), (g_4, n), (g_7, 5)$ and $(g_8, 6)$ form an antichain of maximum size).

9.2 Algorithm RS and its Analysis

In this section we formulate algorithm RS (Random Select), analyze its competitiveness, and present a result on the competitiveness of deterministic algorithms.

9.2.1 Description of Algorithm RS

We consider the natural randomized algorithm RS where a processor (or group) with knowledge that the tasks in a set $K \subset [n]$ have been completed selects to next complete a task at random from the set $[n] \setminus K$. (Recall that we treat randomized algorithms as distributions over deterministic algorithms.) More formally, let $\Pi = (\pi_1, \ldots, \pi_p)$ be a p-tuple of permutations, where each π_i is a permutation of $[n]$. We describe a deterministic algorithm D_Π so that

$$\mathrm{RS} = \mathcal{R}\big(\{\mathrm{D}_\Pi \mid \Pi \in (\mathcal{S}_n)^p\}\big);$$

here $\mathcal{S}_n$ is the collection of permutations on $[n]$. Let G be a group of processors and $q \in G$ the processor in G with the lowest processor identifier. Then the deterministic algorithm D_Π specifies that the group G, should it know that the tasks in $K \subset [n]$ have been completed, next completes the first task in the sequence $\pi_q(1), \ldots, \pi_q(n)$ which is not in K.

9.2.2 Analysis of Algorithm RS

We now analyze the competitiveness (in terms of task-oriented work) of algorithm RS. For a computation template C we write $W_{\mathrm{RS}}(C) = \mathbb{E}\,[W_{\mathrm{RS}}(C)]$, this expectation taken over the random choices of the algorithm. Where C can be inferred from context, we simply write W_{RS} and W_{OPT}.

We first recall Dilworth's Lemma, a duality theorem for posets:

Lemma 9.10. (Dilworth's Lemma) *The width of a poset P is equal to the minimum number of chains needed to cover P. (A family of nonempty subsets of a set Q is said to* cover *Q if their union is Q.)*

We will also use a generalized degree-counting argument:

Lemma 9.11. *Let $G = (U, V, E)$ be an undirected bipartite graph with no isolated vertices and $h : V \to \mathbb{R}$ a non-negative weight function on G. For a vertex v, let $\Gamma(v)$ denote the vertices adjacent to v. Suppose that for some $B_1 > 0$ and for each vertex $u \in U$ we have $\sum_{v \in \Gamma(u)} h(v) \leq B_1$ and that for some $B_2 > 0$ and for each vertex $v \in V$ we have $\sum_{u \in \Gamma(v)} h(u) \geq B_2$, then* $\dfrac{\sum_{u \in U} h(u)}{\sum_{v \in V} h(v)} \geq \dfrac{B_2}{B_1}$.

Proof. We compute the quantity $\sum_{(u,v) \in E} h(u)h(v)$ by expanding according to each side of the bipartition:

$$B_1 \sum_{u \in U} h(u) \geq \sum_{u \in U} \Big(h(u) \cdot \sum_{v \in \Gamma(u)} h(v) \Big) = \sum_{(u,v) \in E} h(u)h(v)$$
$$= \sum_{v \in V} \Big(h(v) \cdot \sum_{u \in \Gamma(v)} h(u) \Big) \geq B_2 \sum_{v \in V} h(v).$$

As $B_1 > 0$ and $\sum_v h(v) \geq B_2 > 0$, we conclude that $\dfrac{\sum_{u \in U} h(u)}{\sum_{v \in V} h(v)} \geq \dfrac{B_2}{B_1}$, as desired. □

We now establish an upper bound on the competitiveness of the algorithm RS.

Theorem 9.12. *Algorithm RS is $(1 + \mathbf{cw}(C)/e)$-competitive for any (p, n)-DAG $C = (V, E)$.*

Proof. Let C be a (p, n)-DAG; recall that associated with C are the two functions $h : V \to \mathbb{N}$ and $\gamma : V \to 2^{[p]} \setminus \{\emptyset\}$. For a subgraph $C' = (V', E')$ of C, we let $H(C') = \sum_{v \in V'} h(v)$. Recall that $P_C(v)$ and $S_C(v)$ denote the predecessor and successor graphs of C at v. Then, we say that a vertex $v \in V$ is *saturated* if $H(P_C(v)) \leq n$; otherwise, v is *unsaturated*. Note that if v is saturated, then the group $\gamma(v)$ must complete $h(v)$ tasks *regardless of the scheduling algorithm used*. Along these same lines, if v is an unsaturated vertex for which $n > \sum_{u<v} h(u)$, the group $\gamma(v)$ must complete at least $\max(h(v), n - \sum_{u<v} h(u))$ tasks under any scheduling algorithm. As these portions of C which correspond to computation which must be performed by any algorithm will play a special role in the analysis, it will be convenient for us to rearrange the DAG so that all such work appears on saturated vertices. To achieve this, note that if v is an unsaturated vertex for which $\sum_{u<v} h(u) < n$, we may replace v with a pair of vertices, v_s and v_u, where all edges directed into v are redirected to v_s, all edges directed out of v are changed to originate at v_u, the edge (v_s, v_u) is added to E, and h is redefined so that

$$h(v_s) = n - \sum_{u<v} h(u) \qquad \text{and} \qquad h(v_u) = h(v) - h(v_s).$$

Note that the graph C' obtained by altering C in this way corresponds to the same computation, in the sense that $W_{\mathrm{D}}(C) = W_{\mathrm{D}}(C')$ for any algorithm D. For the remainder of the proof we will assume that this alteration has been made at every relevant vertex, so that the graph C satisfies the condition

$$v \text{ unsaturated } \Rightarrow \sum_{u<v} h(u) \geq n. \tag{9.1}$$

Finally, for a vertex v, we let T_v be the random variable equal to the number of tasks that RS completes at vertex v. Note that if v is saturated, then $T_v = h(v)$. Let $\mathcal{S}$ and $\mathcal{U}$ denote the sets of saturated and unsaturated vertices, respectively. Given the above definitions, we immediately have

$$W_{\mathrm{OPT}} \geq \sum_{s\in\mathcal{S}} h(s)$$

and, by linearity of expectation,

$$W_{\mathrm{RS}} = \mathbb{E}\Big[\sum_v T_v\Big] = \sum_{s\in\mathcal{S}} h(s) + \sum_{u\in\mathcal{U}} \mathbb{E}[T_u] \leq W_{\mathrm{OPT}} + \sum_{u\in\mathcal{U}} \mathbb{E}[T_u]. \tag{9.2}$$

Our goal is to conclude that for some appropriate β,

$$\mathbb{E}\left[\sum_{u\in\mathcal{U}} T_u\right] \leq \beta \cdot \sum_{s\in\mathcal{S}} h(s) \leq \beta \cdot W_{\mathrm{OPT}}$$

and hence that RS is $1+\beta$ competitive. We will obtain such a bound by applying Lemma 9.11 to an appropriate bipartite graph, constructed next.

Given $C = (V, E)$ construct the (undirected) bipartite graph $G = (\mathcal{S}, \mathcal{U}, E_G)$ where $E_G = \{(s, u) \mid s < u\}$. As in Lemma 9.11, for a vertex v, we let $\Gamma(v)$ denote the set of vertices adjacent to v. Now assign weights to the vertices of G according to the rule $h^*(v) = \mathbb{E}[T_v]$. Note that for $s \in \mathcal{S}, h^*(s) = h(s)$ and hence by condition (9.1) above, we immediately have the bound

$$\forall u \in \mathcal{U}, \quad \sum_{s\in\Gamma(u)} h^*(s) \geq n. \tag{9.3}$$

We now show that $\forall s \in \mathcal{S}$,

$$\sum_{u\in\Gamma(s)} h^*(u) \leq \mathbf{cw}(C) \cdot \frac{n}{e}. \tag{9.4}$$

Before proceeding to establish this bound, note that equations (9.3) and (9.4), together with Lemma 9.11 imply that

$$\begin{aligned} W_{\mathrm{RS}}(C) &\leq \sum_{s\in\mathcal{S}} h(s) + \sum_{u\in\mathcal{U}} h^*(u) \leq \Big(1 + \frac{\mathbf{cw}(C)}{e}\Big) \sum_{s\in\mathcal{S}} h(s) \\ &\leq \Big(1 + \frac{\mathbf{cw}(C)}{e}\Big) W_{\mathrm{OPT}}(C), \end{aligned}$$

as desired.

Returning now to equation (9.4), let $s \in \mathcal{S}$ be a saturated vertex and consider the successor graph (of C) at s, $S_C(s)$. By Lemma 9.10 (Dilworth's Lemma), there exist $w \triangleq \mathbf{w}(S_C(s)) \leq \mathbf{cw}(C)$ paths in $S_C(s)$, $P_1, P_2, \ldots P_w$ so that their union covers $S_C(s)$. Let X_i be the random variable whose value is the number of tasks performed by RS on the portion of the path P_i consisting of unsaturated vertices. Note that if $u \in V$ is unsaturated and $u \leq v$, then v is unsaturated and hence, for each path P_i, there is a first unsaturated vertex u_i^0 after which every vertex of P_i is unsaturated. Note now that for a fixed individual task τ, conditioned upon the event that τ is not yet complete, the probability that τ is *not* chosen by RS for completion at a given selection point in $P_C(u_i^0)$ is no more than $(1 - 1/n)$. Let L_i be the random variable whose value is the set of tasks left incomplete by RS at the formation of the group $\gamma(u_i^0)$. As u_i^0 is unsaturated, $\sum_{v<u_i^0} h(v) \geq n$ by condition (9.1) and hence, for each i,

$$\Pr[\tau \in L_i] \leq (1 - 1/n)^n \leq 1/e.$$

As there are a total of n tasks,

$$\mathbb{E}[|L_i|] \leq n/e.$$

Of course, since RS completes a new task at each step, $X_i \leq |L_i|$ so that $\mathbb{E}[X_i] \leq n/e$ and by the linearity of expectation

$$\mathbb{E}\Big[\sum_i X_i\Big] \leq w \cdot n/e.$$

Now every unsaturated vertex in $S_C(s)$ appears in some P_i and hence

$$\sum_{u \in \Gamma(s)} h^*(u) \leq \mathbb{E}\Big[\sum_i X_i\Big] \leq wn/e \leq \mathbf{cw}(C) \cdot n/e,$$

as desired. □

9.2.3 Deterministic Algorithms

The analysis of algorithm RS can be altered to yield an upper bound result on the competitiveness of deterministic algorithms. Recall that a deterministic algorithm D for the *Omni-Do* problem in this setting is a rule which, given a processor (or group of processors) and a collection of tasks known to be completed, determines the next task for this processor (or group) to complete. Specifically, an algorithm is a function $D : 2^{[p]} \times 2^{[n]} \to [n]$; Furthermore, we assume that $D(P, T) \notin T$ for all $P \subset [p]$ and for all $T \subsetneq [n]$, which is to say that the algorithm never chooses to complete a task it already knows to be completed (thus we restrict our attention to nontrivial algorithms). Then,

Theorem 9.13. *Any (nontrivial) deterministic algorithm D for Do-All$_{\mathcal{A}_{GR}}(n,p,f)$ is $(1+\mathbf{cw}(C))$-competitive for any (p,n)-DAG $C=(V,E)$.*

Proof. In the proof of Theorem 9.12, $h^*(v)$ was defined as the expected number of tasks performed by algorithm RS at node v. For algorithm D, if we define $h^*(v)$ to be the actual number of tasks performed by the algorithm at node v, then it is not difficult to see that equation (9.4) becomes $\sum_{u\in\Gamma(s)} h^*(u) \leq \mathbf{cw}(C)\cdot t$ (provided that no processor in D performs a task that already knows its result). This leads to the thesis of the theorem. □

9.3 Lower Bounds

We now show that the competitive ratio achieved by algorithm RS is tight. We begin with a lower bound for deterministic algorithms. This is then applied to give a lower bound for randomized algorithms in Corollary 9.15.

Theorem 9.14. *Let $a:\mathbb{N}\to\mathbb{R}$ and D be a deterministic algorithm for Omni-Do so that D is $a(\mathbf{cw}(\cdot))$-competitive (that is D is α-competitive, for a function $\alpha = a\circ\mathbf{cw}$)). Then $a(c)\geq 1+c/e$.*

Proof. Fix $k\in\mathbb{N}$. Consider the case when $n=p=g\gg k$ and $n \bmod k = 0$, g being the number of initial groups. We consider a computation template $C_{\mathbf{G}}$ determined by a tuple $\mathbf{G}=(G_1,\ldots,G_{n/k})$ where each $G_i\subset[n]$ is a set of size k and $\bigcup_i G_i=[n]$. Initially, the computation template $C_{\mathbf{G}}$ has the processors synchronously proceed until each has completed n/k tasks; at this point, the processors in G_i are merged and allowed to exchange information about task executions. Each G_i is then immediately partitioned into c groups. Note that the off-line optimal algorithm accrues exactly n^2/k work for this computation template (it terminates prior to the partitions of the G_i).

We will show that for any D, there is a selection of the G_i so that

$$W_{\mathrm{D}}(C_{\mathbf{G}}) \geq n^2/k\left[1+c\left(1-\frac{1}{k}\right)^k - o(1)\right],$$

and hence that $a(c)\geq 1+c/e$. Consider the behavior of D when the $\mathbf{G}$ is selected at random, uniformly among all such tuples. Let $P_i\subset[n]$ be the subset of n/k tasks completed by processor i before the merges take place; these sets are determined by the algorithm D. We begin by bounding

$$\mathbb{E}_{\mathbf{G}}\left[\left|\bigcup_{i\in G_1} P_i\right|\right].$$

To this end, consider an experiment where we select k sets $Q_1,\ldots,Q_k$, each Q_i selected independently and uniformly from the set $\{P_i\}$. Now, for a specific task τ, let $p_\tau = \Pr_{Q_1}[\tau\notin Q_1]$, so that $\Pr_{Q_i}[\tau\notin\bigcup_i Q_i] = p_\tau^k$. As the Q_i are selected independently,

$$\mathbb{E}_{Q_i}\Big[\big|[n]-\bigcup_i Q_i\big|\Big]=\sum_\tau p_\tau^k.$$

Observe now that

$$\sum_\tau(1-p_\tau)=\sum_\tau \Pr_{Q_1}[\tau\in Q_1]=\mathbb{E}_{Q_1}[|Q_1|]=n/k$$

and hence $\sum_\tau p_\tau = n(1-1/k)$. As the function $x\mapsto x^k$ is convex on $[0,\infty)$, $\sum_\tau p_\tau^k$ is minimized when the p_τ are equal and we must have

$$\mathbb{E}_{Q_i}\Big[\big|[n]-\bigcup_i Q_i\big|\Big]\geq n\cdot\left(1-\frac{1}{k}\right)^k.$$

Now observe that, conditioned on the Q_i being distinct, the distribution of $(Q_1,\ldots,Q_k)$ is identical to that of $(P_{g_1^1},\ldots,P_{g_k^1})$ where the random variable $G_1=\{g_1^1,\ldots,g_k^1\}$. Considering that $\Pr[\exists i\neq j, Q_i=Q_j]\leq k^2/n$, we have

$$\mathbb{E}_{Q_i}\Big[\big|[n]-\bigcup_i Q_i\big|\Big]\leq\left(1-\frac{k^2}{n}\right)\mathbb{E}_{\mathbf{G}}\Big[n-\big|\bigcup_{i\in G_1}P_i\big|\Big]+1\cdot\frac{k^2}{n}$$

and hence as $n\to\infty$ we see that the expected number of tasks remaining for those processors in group G_1 is

$$\mathbb{E}_{\mathbf{G}}\left[n-\Big|\bigcup_{i\in G_1}P_i\Big|\right]\geq n(1-1/k)^k-o(1).$$

Of course, the distribution of each G_i is the same, so that

$$\mathbb{E}_{\mathbf{G}}\left[\sum_{i=1}^{n/k}\Big(n-\big|\bigcup_{j\in G_i}P_j\big|\Big)\right]=[1-o(1)]\left(\frac{n}{k}\right)\cdot n\left(1-\frac{1}{k}\right)^k.$$

In particular, there must exist a specific selection of $\mathbf{G}=(G_1,\ldots,G_{n/k})$ which achieves this bound. Recall that every G_i is partitioned into c groups. Therefore, for such $\mathbf{G}$, the total work is at least

$$\frac{n^2}{k}\cdot\left(1+[1-o(1)]\cdot c\cdot\left(1-\frac{1}{k}\right)^k\right).$$

As $\lim_{k\to\infty}(1-\frac{1}{k})^k=\frac{1}{e}$, this completes the proof. □

The above lower bound result together with the upper bound result given in Theorem 9.13 show that there is a gap of a factor of $1/e$ on the competitiveness of deterministic algorithms. Closing this gap remains an open problem.

As the above stochastic computation template $C_{\mathbf{G}}$ is independent of the deterministic algorithm D, this immediately gives rise to a lower bound for randomized algorithms:

Corollary 9.15. *Let $\mathcal{R}(\{D_\zeta \mid \zeta \in Z\})$ be a randomized algorithm for Omni-Do that is $(a \circ \mathbf{cw})$-competitive, where $a : \mathbb{N} \to \mathbb{R}$. Then $a(c) \geq 1 + c/e$.*

Proof. Assume for contradiction that for some c, $a(c) < 1 + c/e$ and let k be large enough so that $(1 - \frac{1}{k})^k > a(c) - 1$. For this k we proceed as in the proof above, considering a random $\mathbf{G}$ and the computation template $C_{\mathbf{G}}$ with $n = g = p$ congruent to 0 mod k, g being the number of initial groups. Then, as above,

$$\mathbb{E}_{\mathbf{G}}\left[\mathbb{E}_{\zeta}\left[W_{\mathrm{D}_\zeta}(C_{\mathbf{G}})\right]\right] = \mathbb{E}_{\zeta}\left[\mathbb{E}_{\mathbf{G}}\left[W_{\mathrm{D}_\zeta}(C_{\mathbf{G}})\right]\right] \geq \min_{\zeta}\left[\mathbb{E}_{\mathbf{G}}\left[W_{\mathrm{D}_\zeta}(C_{\mathbf{G}})\right]\right]$$
$$\geq \frac{n^2}{k} \cdot \left(1 + [1 - o(1)] \cdot c \cdot \left(1 - \frac{1}{k}\right)^k\right).$$

Hence there exists a $\mathbf{G}$ so that $\mathbb{E}_{\zeta}\left[W_{\mathrm{D}_\zeta}(C_{\mathbf{G}})\right] \geq \frac{n^2}{k} \cdot \left(1 + [1 - o(1)]\frac{c}{e}\right)$, which completes the proof. □

The above result yields the optimality of algorithm RS. Specifically, RS achieves the optimal competitive ratio over the set of all computation templates with a given computation width.

9.4 Open Problems

One outstanding open question is how to derandomize the schedules used by task-performing algorithms in this chapter. Specifically, we would like to construct deterministic scheduling algorithms that are $(1+\mathbf{cw}(C)/e)$-competitive for any computation template C, thus closing the gap of factor $1/e$ identified in the previous section.

An interesting direction is to study the competitiveness of *Omni-Do* algorithms with respect to their message complexity. Another promising direction is to study the task-performing paradigm in the models of computation that combine network regroupings with processor failures. The goal is to establish complexity results that show how performance of task-performing algorithms depends both on the extent of regroupings and on the number of processor failures.

9.5 Chapter Notes

Dolev, Segala, and Shvartsman [29] performed the first study of the *Omni-Do* problem in the partitionable setting. Assuming $p = n$, they model regrouping patterns for which the termination time of any on-line task-performing algorithm is greater than the termination time of an off-line task-performing algorithm by a factor linear in p.

Malewicz, Russell, and Shvartsman [83, 86] introduced the notion of k-*waste* that measures the worst-case redundant work performed by k groups (or processors) when started in isolation and merged into a single group at some later time. They developed several efficient constructions that allow processors to compute locally, without coordination, while controlling waste. These results are deterministic, and they adequately describe such computation to the point of the first regrouping, where the regrouping is assumed to merge groups. (This is the topic of the next Chapter.)

Georgiou and Shvartsman [48] give upper bounds on work for an algorithm that performs work in the presence of network fragmentations and merges using a group communication service where processors initially start in a single group (this is the topic of Chapter 8). They establish an upper bound of $O(\min(n \cdot p,\ n + n \cdot g(C)))$ onw work, where $g(C)$ is the total number of new groups formed during the computation pattern C. Note that $\mathbf{cw}(C) \leq g(C)$, and there can be an arbitrary gap between $\mathbf{cw}(C)$ and $g(C)$.

The presentation in this chapter is based on the work of Georgiou, Russell, and Shvartsman [46]. For a proof of the Dilworth's lemma see [26].

The notion of competitiveness was introduced by Sleator and Tarjan [105]. See also Bartal, Fiat, and Rabani [11], Awerbuch, Kutten, and Peleg [8], and Ajtai, Aspnes, Dwork, and Waarts [3].