10

Cooperation in the Absence of Communication

IN the setting where the *Omni-Do* (and *Do-All*) problem needs to be solved by distributed message-passing processors there exists a trade-off between computation and communication: both resources must be managed to decrease redundant computation and to ensure efficient computational progress. In this chapter we specifically examine the extreme situation of collaboration *without communication*. That is, we consider the extent to which efficient collaboration is possible if all resources are directed to computation at the expense of communication. Of course there are also cases where such an extreme situation is not a matter of choice: the network may fail, the mobile nodes may have intermittent connectivity, and when communication is unavailable it may take a long time to (re)establish connectivity. The results summarized in this section precisely characterize the ability of distributed agents to collaborate on a known collection of independent tasks by means of local scheduling decisions that require no communication and that achieve low redundant work in task executions. Such scheduling solutions exhibit an interesting connection between the distributed collaboration problem and the mathematical design theory. The lower bounds presented here along with the randomized and deterministic schedule constructions show the limitations on such low-redundancy cooperation and show that schedules with near-optimal redundancy can be efficiently constructed by processors working in isolation.

Let us consider an asynchronous setting, where processors communicate by means of a *rendezvous*, i.e., two processors that are able to communicate can perform state exchange. The processors that are not able to communicate via rendezvous have no choice but to perform all n tasks. Consider the computation with a single rendezvous. There are $p-2$ processors that are unable to communicate, and they collectively must perform exactly $n \cdot (p-2)$ work units to learn all results. Now what about the remaining pair of processors that are able to rendezvous? In the worst case they rendezvous after performing all tasks individually. In this case no savings in work are realized. Suppose they rendezvous having performed $n/2$ tasks each. In the best case, the two processors performed mutually-exclusive subsets of tasks and they learn the

complete set of results as a consequence of the rendezvous. In particular if these two processors know that they will be able to rendezvous in the future, they could schedule their work as follows: one processor performs the tasks in the order $1, 2, \ldots, n$, the other in the order $n, n-1, \ldots, 1$. No matter when they happen to rendezvous, the number of tasks they both perform is minimized. Of course the processors do not know *a priori* what pair will be able to rendezvous. Thus it is interesting to produce task execution schedules for all processors, such that upon the first rendezvous of any two processors the number of tasks performed redundantly is minimized.

This setting we have just described is interesting for several reasons. If the communication links are subject to failures, then each processor must be ready to execute all of the n tasks, whether or not it is able to communicate. In realistic settings the processors may not initially be aware of the network configuration, which would require expenditure of computation resources to establish communication, for example in radio networks. In distributed environments involving autonomous agents, processors may *choose* not to communicate either because they need to conserve power or because they must maintain radio silence. Finally, during the initial configuration of a dynamic network or a middleware service (such as a group communication service) the individual processors may start working in isolation pending the completion of system configuration. Regardless of the reasons, it is important to direct any available computation resources to performing the required tasks as soon as possible. In all such scenarios, the n tasks have to be scheduled for execution by all processors. The goal of such scheduling must be to control redundant task executions in the absence of communication and during the period of time when the communication channels are being (re)established.

Chapter structure.

In Section 10.1 we describe the adverse setting, formalize the notions of schedules, waste associated with redundant task execution in schedules, and present basic design theory. In Section 10.2 we present a lower bound on redundancy without communication. Section 10.3 explores the behavior of random schedules. Derandomization of schedules is the topic of Section 10.4. Discussion of open problems is in Section 10.5.

10.1 Adversity, Schedules, Waste, and Designs

The adversarial setting. In our abstract setting there are p asynchronous processors that need to perform n tasks. The processors have unique identifiers from the set $[p] = \{1, \ldots, p\}$, and the tasks have unique identifiers from the set $[n] = \{1, \ldots, n\}$. Initially each processor knows the tasks that need to be performed and their identifiers (otherwise no fault-tolerant distributed solution is possible). For this setting, the adversary initially isolates the processors, which forces them to perform tasks without being able to coordinate

their activity with other processors. The adversary then allows the processors to rendezvous, but with the goal of maximizing the redundant work performed by the processors prior to the rendezvous.

For the purposes of this chapter, we define a simplified adversary, called $\mathcal{A}_R$, that starts processors in isolation, and then causes a rendezvous. We also define a parameterized adversary $\mathcal{A}_R^{(r)}$ to be the adversary that causes at most a r-way rendezvous. Following our established notation, for an algorithm A, let $\mathcal{E} = \mathcal{E}(A, \mathcal{A}_R^{(r)})$ be the set of all executions of the algorithm in our model of computation subject to adversary $\mathcal{A}_R^{(r)}$. For a particular execution $\xi \in \mathcal{E}$, the adversarial pattern $\xi|_{\mathcal{A}_R^{(r)}}$ establishes that the processors $q_1, \ldots, q_k$, where $k \leq r$, rendezvous for the first time when each processor q_i performs a_1 tasks prior to the rendezvous. Note that each a_i can be very different due to asynchrony. We define the *weight* $||\xi|_{\mathcal{A}_R^{(r)}}||$ of the adversarial pattern corresponding to this execution to be the vector $\boldsymbol{a} = (a_1, \ldots, a_k)$.

We are interested in studying how the magnitude of the redundant work depends on the weight of the adversarial pattern.

Schedules and waste. A (p, n)-*schedule* is a tuple $(\sigma_1, \ldots, \sigma_p)$ of p permutations of the set $[n]$. When $p = 1$ it is elided and we simply write n-*schedule*. A (p, n)-schedule immediately gives rise to a strategy for p isolated processors who must complete n tasks until communication between some pair (or group) is established: the processor i simply proceeds to complete the tasks in the order prescribed by σ_i. Suppose now that an adversarial pattern causes some k of these processors, say $q_1, \ldots, q_k$, to rendezvous at a time when the ith processor in this group, q_i, has completed a_i tasks (i.e., the weight of the corresponding adversarial pattern is $\boldsymbol{a} = (a_1, \ldots, a_k)$). Ideally, the processors would have completed disjoint sets of tasks, so that the total number of tasks completed is $\sum_i a_i$. As this is too much to hope for in general, it is natural to attempt to bound the gap between $\sum_i a_i$ and the actual number of distinct tasks completed. This gap we call *waste* (here and throughout, if $\phi : X \to Y$ is a function and $L \subset X$, we let $\phi(L) = \{\phi(x) \mid x \in L\}$):

Definition 10.1. *If L is a (p, n)-schedule and $(a_1, \ldots, a_k) \in \mathbb{N}^k$, the* ***waste*** *function for L is*

$$\mathfrak{W}_L(a_1, \ldots, a_k) = \max_{(q_1, \ldots, q_k)} \left(\sum_i^k a_i - \left| \bigcup_i^k \sigma_{q_i}([a_i]) \right| \right),$$

this maximum taken over all k tuples $(q_1, \ldots, q_k)$ of distinct elements of $[p]$.

For a specific vector $\boldsymbol{a} = (a_1, \ldots, a_k)$ representing the weight of an adversarial pattern, $\mathfrak{W}_L(\boldsymbol{a})$ captures the worst-case number of redundant tasks performed by any collection of k processors when the ith process has completed the first a_i tasks of its schedule.

One immediate observation is that bounds on *pairwise* waste can be naturally extended to bounds on *k-wise* waste: specifically, note that if L is a (p, n)-schedule then

$$\mathfrak{W}_L(a_1, \ldots, a_k) \leq \sum_{i<j} \mathfrak{W}_L(a_i, a_j)$$

just by considering the first two terms of the standard inclusion-exclusion rule. Moreover, it appears that this relationship is fairly tight as it is nearly attained by randomized schedules (see Section 10.3). With this justification we shall content ourselves to focus mainly on pairwise waste—the function $\mathfrak{W}_L(a, b)$.

Designs as schedules. Set systems with prescribed intersection properties have been the object of intense study by both the design theory community and the extremal set theory community. Despite this, the study of *waste* in distributed cooperative settings is new. We shall, however, make substantial use of some design-theoretic constructions, which we describe below.

Definition 10.2. *A* ℓ-(v, k, λ) design *is a family of subsets* $\mathcal{L} = (L_1, \ldots, L_t)$ *of the set* $[v]$ *with the property that each* $|L_i| = k$ *and any set of* ℓ *elements of* $[v]$ *is a subset of precisely* λ *of the* L_i*. (N.B. The subsets* L_i *are typically referred to as* blocks*.)*

Observe that if $\mathcal{L}$ is a ℓ-(v, k, λ) design, then it is also a $(\ell - 1)$-$(v, k, \hat{\lambda})$ design where

$$\hat{\lambda} = \lambda \frac{(v - \ell + 1)}{(k - \ell + 1)}.$$

To see this, note that if T is a subset of elements of size $\ell - 1$, then there are exactly $v - (\ell - 1)$ sets of size ℓ which contain T; let $U_i, i \in [v - (\ell - 1)]$, denote these sets. By assumption, each U_i appears in exactly λ of the L_j. Of course, if U_i is a subset of some L_j, then in fact exactly $k - (\ell - 1)$ if the U_i are subsets of L_j. Hence T appears in exactly $\lambda(v - \ell + 1)/(k - \ell + 1)$ of the L_j, as desired.

To see the connection between such designs and our problem, let $\mathcal{D}$ be a 2-(p, k, λ) design consisting of n sets $L_1, \ldots, L_n$. For each $i \in [p]$, let $T_i = \{j \mid i \in L_j\}$. Note now that for any $i \neq j$,

$$T_i \cap T_j = \{k \mid \{i, j\} \subset L_k\}$$

and hence that $|T_i \cap T_j| = \lambda$. Based on the observation above, we see also that $\forall i, j, |T_i| = |T_j|$ and let a denote this common cardinality. Now, let $\Sigma = (\sigma_1, \ldots, \sigma_t)$ be any sequence of permutations of $[n]$ for which $\sigma_i([a]) = T_i$. It is clear that these form an (p, n)-schedule for which

$$\mathfrak{W}_\Sigma(a, a) = \lambda.$$

Unfortunately, the above construction offers satisfactory control of 2-waste only for the specific pair (a, a). Furthermore, considering that the construction only determines the *sets* $\sigma_i([a])$ and $\sigma_i([p] \setminus [a])$, the ordering of these can be conspiratorially arranged to yield poor bounds on waste for other values. Our goal is construct schedules with satisfactory control on waste for all pairs (a, b).

While designs do not appear to immediately induce a solution to this problem, we will apply the following design-theoretic construction several times in the sequel. Let $\mathrm{GF}(q)$ denote the finite field with q elements, where q is a prime power. Treating $\mathrm{GF}(q)^3$ as a vector space over $\mathrm{GF}(q)$, the design will be given by the lattice of linear subspaces of $\mathrm{GF}(q)^3$. It is easy to check that there are $t = q^2 + q + 1$ distinct one dimensional subspaces of $\mathrm{GF}(q)^3$, which we denote $\ell_1, \ldots, \ell_t$. We say that two subspaces ℓ_i and ℓ_j are *orthogonal* if $\forall u \in \ell_1, \forall v \in \ell_2$, $\langle u, v\rangle = \sum u_j v_j \bmod q = 0$; in this case we write $\ell_i \perp \ell_j$. It is a fact that for any one dimensional subspace there are exactly $q + 1$ one dimensional subspaces to which it is orthogonal. The design consists of the $n = q^2 + q + 1$ sets $S_u = \{\ell_i \mid \ell_i \perp \ell_u\}$. It is easy to show that any pair of such sets intersect at a single ℓ_i, and that this forms a 2-$(q^2 + q + 1, q + 1, 1)$ design.

For concreteness, we fix a specific (arbitrary) ordering of each of these sets L_u: let K_u denote a canonical sequence $\langle k_u^1, \ldots, k_u^r\rangle$ where $L_u = \{\ell_{k_u^i} \mid 1 \le i \le q + 1\}$; i.e., the one dimensional subspaces $\ell_{k_u^i}$, $i = 1, \ldots, q + 1$, are precisely those orthogonal to ℓ_u. For convenience, for two sequences A and B, we let $A \cap B$ and $A \cup B$ denote the corresponding union or intersection of the sets of objects in the sequences. We record the above discussion in the following proposition.

Proposition 10.3. *Let $t = q^2 + q + 1$, where q is a prime power. Then the sequences $\mathcal{K}_t = \langle K_1, \ldots, K_t\rangle$ possess the following properties: each K_u has length $q + 1$, for each $u \neq v$, $|K_u \cap K_v| = 1$, and any element appears in exactly $q + 1$ distinct sequences. We note also that if q is prime, the first element of each sequence can be calculated in $O(\log t)$ time; each subsequent element can be calculated in $O(1)$ time.*

In the sequel we will use these designs with $t = p$, the number of processors. We assume throughout that addition or multiplication of two $\log(\max\{p, n\})$-bit numbers can be performed in $O(1)$ time.

10.2 Redundancy without Communication: a Lower Bound

Controlling global computation redundancy in the absence of communication is a futile task. This is because no amount of algorithmic sophistication can compensate for the possibility of individual processors, or groups of processors,

becoming disconnected during the computation. In general, an adversary that is able to partition the processors into g groups that cannot communicate with each other will cause any task-performing algorithm to have work $\Omega(n \cdot g)$, even if each group of processors performs no more than the optimal number of $\Theta(n)$ tasks. In the extreme case where all processors are isolated from the beginning, the work of any algorithm is $\Omega(n \cdot p)$, which is at least the work of an oblivious algorithm, where each processor performs all tasks.

Of course it is not surprising that substantial redundancy cannot be avoided in the absence of communication, furthermore, the lower bound on work of $\Omega(n \cdot p)$ is not very interesting. However, as we pointed out earlier, it is possible to schedule the work of a pair of processors so that each can perform up to $n/2$ tasks without a single task performed redundantly. Thus it is very interesting to consider the intersection properties of pairs of processor schedules, i.e., 2-waste.

If we insist that among the p total processors, any two processors, having executed the same number of tasks n', where $n' < n$, perform *no* redundant work, then it must be the case that $n' \leq \lfloor n/p \rfloor$. In particular, if $p = n$, then the pairwise waste jumps to one if any processor executes more than one task. The next natural question is: how many tasks can processors complete before the lower bound on pairwise redundant work is 2? In general, if any two processors perform n_1 and n_2 tasks respectively, what is the lower bound on pairwise redundant work? In this section we answer these questions. The answers contain both good and bad news: given a fixed t, the lower bound on pairwise redundant work starts growing slowly for small n_1 and n_2, then grows quadratically in the schedule length as n_1 and n_2 approach t.

Now we proceed to the lower bound for the case when two processors execute *different* number of tasks prior to their rendezvous (this lower bound generalizes the second Johnson Bound).

Theorem 10.4. *Let $\Pi = \langle \pi_1, \ldots, \pi_p \rangle$ be a (p, n)-schedule and let $0 \leq a \leq b \leq n$. Then*

$$\mathfrak{W}_\Pi(a, b) \geq \frac{p\ a^2}{(p-1)(n-b+a)} - \frac{a}{p-1}.$$

For example, when processors perform the same number of tasks $a = b$ and $p = n$, then the worst case number of redundant tasks for any pair is at least $\frac{a^2-a}{n-1}$. This means that (for $p = n$) if a exceeds $\sqrt{n}+1$, then the number of redundant task is at least 2.

Corollary 10.5. *For $n = p$, if $a > \sqrt{n - 3/4} + \frac{1}{2}$ then any p-processor schedule of length a for n tasks has worst case pairwise waste at least 2.*

10.3 Random Schedules

As one would expect, schedules chosen at random perform quite well. In this section we explore the behavior of the (p, n)-schedules obtained when each

permutation is selected uniformly (and independently) at random among all permutations of $[n]$.

Randomized schedules

When the processors are endowed with a reasonable source of randomness, a natural candidate scheduling algorithm is one where processors select tasks by choosing them uniformly among all tasks they have not yet completed. This amounts to the selection, by each processor i, of a random permutation $\pi_i \in \mathcal{S}_{[n]}$ which determines the order in which this processor will complete the tasks. ($\mathcal{S}_{[n]}$ denotes the collection of all permutations of the set $[n]$.) We let $\mathcal{R}$ be the resulting system of schedules.

Our objective now is to show that random schedules $\mathcal{R}$ have controlled waste with high probability. This amounts to bounding, for each pair i, j and each pair of numbers a, b, the overlap $|\pi_i([a]) \cap \pi_j([b])|$. Observe that when these π_i are selected at random, the expected size of this intersection is ab/n. By showing that the actual waste is very likely to be close to this expected value, one can conclude the waste if bounded for *all* long enough prefixes.

Theorem 10.6. *Let $\mathcal{R}$ be a system of p random schedules for n tasks constructed as above. Then with probability at least $1 - \frac{1}{pn}$, $\forall a, b$ such that $7\sqrt{n}\ln(2pn) \leq a, b \leq n$, $\mathfrak{W}_{\mathcal{R}}(a, b) \leq \dfrac{ab}{n} + \Delta(a, b)$, where $\Delta(a, b) = 11\sqrt{\frac{ab}{n}\ln(2pn)}$.*

Observe that Theorem 10.4 shows that (p, n)-schedules must have waste $\mathfrak{W}(a, a) = \Omega(a^2/n)$ (as $p \to \infty$); hence such randomized schedules offer nearly optimal waste for this case.

k-Waste for random schedules

For random schedules, one can apply martingale techniques to directly control k-wise waste. We mention one such result.

Theorem 10.7. *Consider the random schedule $\mathcal{R}$ as given above. Then with probability at least $1 - 1/p$,*

$$\mathfrak{W}_{\mathcal{R}}(a, \ldots, a) \leq \sum_{s=2}^{k} (-1)^s \binom{k}{s} \frac{a^s}{n^{s-1}} + \Delta_{a,k},$$

where $\Delta_{a,k} = (2k+1)\sqrt{a \ln p}$.

Note that again this bounds the distance of the k-waste from its expected value, which can be computed by inclusion-exclusion to be $\sum_{s=2}^{k} (-1)^s \binom{k}{s} \frac{a^s}{n^{s-1}}$. The proof, which we omit, proceeds by considering the martingale which exposes the ith element of all schedules at step i. The theorem then follows by noting that the expected value can change by at most k during a single exposure and applying Azuma's inequality.

10.4 Derandomization via Finite Geometries

We now consider a method for derandomizing these schedules using the design discussed in Section 10.1.

Schedules for $p = n$

We construct a system of schedules of length p by arranging tasks from the sequences of $\mathcal{K}_p$ in a recursive fashion. (Recall that while the sequences of $\mathcal{K}_p$ have strong intersection properties, they are only roughly $\sqrt{p}$ in length.) In preparation for the recursive construction, we record the following lemma about the pairwise intersections of the elements in the sequence of $\mathcal{K}_p$ *indexed* by a specific subspace K_u.

Lemma 10.8. *Let $\mathcal{K}_p = \langle K_1, \ldots, K_p \rangle$ be the collection of sequences constructed in Proposition 10.3, and let $K_u = \langle k_u^1, \ldots, k_u^{q+1} \rangle$, $1 \le u \le p$. Then for any $i \neq j$, we have $K_{k_u^i} \cap K_{k_u^j} = \{u\}$.*

As a result of this lemma, there is *only a single repeated element* in the sequences $K_{k_u^1}, K_{k_u^2}, \ldots, K_{k_u^{q+1}}$; this element is u. This fact suggests the following construction of a system of schedules $\mathcal{Q}_p$. Let Q_u, $1 \le u \le p$, be the sequence whose first element is u, and whose remaining elements are given by concatenating the $q+1$ sequences $K_{k_u^1}, \ldots, K_{k_u^{q+1}}$ after removing u from each. Specifically,

$$Q_u = \langle u \rangle \circ (\bigcirc_{i \in K_u} (K_i - u)),$$

where $\circ$ denotes concatenation and $K_i - u$ denotes the sequence K_i with u deleted. Note now that since the total length of Q_u is evidently $(q+1)q+1 = p$, each element of $[p]$ must appear exactly once in each Q_u; these Q_u thus give rise to a family of permutations π_u, where $\pi_u(i)$ is the ith element of Q_u. Let $\mathcal{Q}_p = (\pi_1, \ldots, \pi_p)$.

We conceptually divide the sequences Q_u (associated with the permutations π_u) into $q+1$ *segments* of elements. The first segment contains the first $q+1$ elements (including the initial element u); the remaining q segments contain q consecutive elements each.

This recursive construction yields a straightforward bound on pairwise waste, recorded below.

Theorem 10.9. *Let q be a prime power, $p = q^2 + q + 1$. Let $a = 1 + iq$, $b = 1 + jq$, $0 \le i, j \le q+1$. Then*

$$\mathfrak{W}_{\mathcal{Q}_p}(a,b) \le \begin{cases} 0, & i + j = 0, \\ 1, & i = 0, j \ge 1 \text{ or } i \ge 1, j = 0, \\ q + ij, & i \cdot j \ge 1. \end{cases}$$

We mention that the construction can be done on-line. For each schedule the first element can be calculated in $O(1)$ time. For the remaining $q(q+1)$ elements, at the beginning of every sequence of q elements we need to invert at most two elements in $\mathrm{GF}(q)$. When q is prime this can be done in $O(\log p)$ using the extended Euclidean algorithm. Other elements of the schedule can be found in $O(1)$ time.

Note that when $n = \kappa p$ for some $\kappa \in \mathbb{N}$, the above construction can be trivially applied by placing the n tasks into p chunks of size κ. In this case, of course, when a single overlap occurred in the original construction, this penalty is amplified by κ.

Controlling waste for short prefixes

One disadvantage of $\mathcal{Q}_p$ is that the first segment may repeat, so that $(q+1)$ waste may be incurred when a prefix of length $\hat{a} = (q+1)$ is executed. To postpone this increase one would like to rearrange the segments in each Q_u so that the first segment is distinct across the resulting schedules. This can be accomplished by finding a bijection $\rho : [p] \to [p]$ such that the sequence K_u contains task $\rho(u)$. (In other words ℓ_u must be orthogonal to $\ell_{\rho(u)}$.) This bijection can then be used to select distinct segments as the first segments of schedules in $\mathcal{Q}_p$.

Consider the bipartite graph $G_p = (U_p, V_p, E_p)$ where $U_p = V_p = [p]$ and $p = q^2 + q + 1$; here q is a prime power. Both U_p and V_p can be placed in one-to-one correspondence with the one dimensional subspaces of $\mathrm{GF}(q)^3$. An edge is placed between $\ell_u \in U_p$ and $\ell_v \in V_p$ when they are orthogonal. Based on the structure of $\mathrm{GF}(q)^3$, it is not hard to show that G_p is $(q+1)$-regular. By Hall's theorem, there is always a perfect matching in a d-regular bipartite graph and note that such a matching yields a permutation ρ with the desired properties. In particular if the edge (u, v) appears in the perfect matching, then we put $\rho(u) = v$. This matching can be found using the Hopcroft-Karp algorithm that runs in time $O(\sqrt{|U| + |V|} \cdot |E|) = O(p^2)$.

We use ρ to construct the system of schedules $\mathcal{G}_p$ such that the first segments are distinct. Specifically, given $\mathcal{K}_p$, the system of schedules $\mathcal{G}_p = \langle \gamma_1, \ldots, \gamma_p \rangle$ is defined as follows. For any $1 \le u \le p$, the sequence G_u is given by

$$G_u = \langle u \rangle \circ (K_{\rho(u)} - \{u\}) \circ (\bigcirc_{i \in K_u - \rho(u)} (K_i - u)).$$

Then γ_u is the permutation associated with G_u.

Theorem 10.10. *Let q be a prime power, $p = q^2 + q + 1$. Let $a = 1 + iq$, $b = 1 + jq$, $0 \le i, j \le q + 1$. Then:*

$$\mathfrak{W}_{\mathcal{G}_p}(a, b) \le \begin{cases} 0, & i + j = 0, \\ 1, & i = 0, j \ge 1 \text{ or } i \ge 1, j = 0, \\ 1, & i \cdot j = 1, \\ q + ij, & i \cdot j > 1. \end{cases}$$

Observe that this construction is time-optimal as it produces p^2 elements and runs in $O(p^2)$ time. However, the algorithm requires $O(p^2)$ time to construct even a single permutation.

10.5 Open Problems

We surveyed results that characterize the ability of p isolated processors to collaborate on a common known set of n tasks. The good news is that the isolated processors can deterministically construct schedules locally, equipped only with the knowledge of n, p, and their unique processor identifiers in $[p]$. Moreover, the cost of constructing such schedules can be largely amortized over the performance of tasks. It is nevertheless interesting to seek more efficient constructions and deterministic constructions that help control k-waste. Although the lower bounds on wasted work mandate that waste must grow quadratically with the number of executed tasks (from 1 to n), such schedules control wasted work for surprisingly long prefixes of tasks. Another worthwhile problem is to design deterministic strategies that control waste for arbitrary patterns of rendezvous, for example, as in the setting of Chapter 9. Finally, for the settings where communication is deemed expensive or undesirable, it is interesting to develop algorithmic and scheduling strategies that intentionally force processors to work in isolation, and to analyze these strategies in terms of waste, work, and message complexity.

10.6 Chapter Notes

The material in this chapter is based on the work of Malewicz, Russell, and Shvartsman [83, 84, 85, 86] and follows the presentation in [99]. The proofs of the theorems and lemmas stated in this Chapter can be found in [86]. Additional results in this area can be found in Malewicz's thesis [81].

The problem of assessing redundant work for distributed cooperation in the absence of communication was studied by Dolev, Segala, and Shvartsman in [29]. The authors showed that for the case of dynamic changes in connectivity, the termination time of any on-line task assignment algorithm can be greater than the termination time of an off-line task assignment algorithm by a factor linear in n. This means that an on-line algorithm may not be able to do better than the trivial solution that incurs linear overhead by having each processor perform all the tasks. With this observation [29] develops an effective strategy for managing the task execution redundancy and proves that the strategy provides each of the $p \leq n$ processors with a schedule of $\Theta(n^{1/3})$ tasks such that at most one task is performed redundantly by any two processors.

Other approaches to dealing with limited communication have also been explored. Papadimitriou and Yannakakis [95] study how limited patterns of

communication affect load-balancing. They consider a problem where there are 3 agents, each of which has a job of a size drawn uniformly at random from $[0, 1]$, and this distribution of job sizes is known to every agent. Any agent A can learn the sizes of jobs of some other agents as given by a directed graph of three nodes. Based on this information each agent has to decide to which of the two servers its job will be sent for processing. Each server has capacity 1, and it may happen that when two or more agents decide to send their jobs to the same server the server will be overloaded. The goal is to devise cooperative strategies for agents that will minimize the chances of overloading any server. The authors present several strategies for agents for this purpose. They show that adding an edge to a graph can improve load balancing. These strategies depend on the communication topology. This problem is similar to our scheduling problem. Sending a job to server number $x \in \{0, 1\}$ resembles doing task number x in our problem. The goal to avoid overloading servers resembles avoiding overlaps between tasks. The problem of Papadimitriou and Yannakakis is different because in our problem we are interested in structuring job execution where the number of tasks can be arbitrary $n \geq 1$.

Georgiades, Mavronicolas, and Spirakis [42] study a similar load-balancing problem. On the one hand their treatment is more general in the sense that they consider arbitrary number of agents n, and arbitrary computable decision algorithms. However it is more restrictive in the sense that they consider only one type of communication topology where there is no communication between processors whatsoever. The two servers that process jobs have some given capacity that is not necessarily 1. They study two families of decision algorithms: algorithms that cannot see the size of jobs before making a decision which server to send a job to for processing, and algorithms that can make decisions based on the size of the job. They completely settle these cases by showing that their decision protocols minimize the chances of overloading any server.

For additional information on the design theory and the extremal set theory see the survey of Hughes and Piper [60]. See [64] for information about the second Johnson Bound. For a discussion of discrete exposure martingales and Azuma's inequality see Alon and Spencer [4]. For Hall's theorem see, e.g., Harary [54]. For Hopcroft-Karp algorithm see [58].