Distributed Computation Meets Design Theory: Local Scheduling for Disconnected Cooperation*

Alexander Russell† Alexander Shvartsman‡

Abstract

Ability to cooperate on common tasks in a distributed setting is key to solving a broad range of computation problems ranging from distributed search such as SETI to distributed simulation and multi-agent collaboration. In such settings there exists a trade-off between computation and communication: both resources must be managed to decrease redundant computation and to ensure efficient computational progress. This survey deals with scheduling issues for distributed collaboration. Specifically, we 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 here 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 redundancy 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. We also show that when processors start working in isolation and are subjected to an arbitrary pattern of network reconfigurations, e.g., fragmentations and merges, randomized scheduling is competitive compared to an optimal algorithm that is aware of the pattern of reconfigurations.

*This article appears as: A. Russell and A. Shvartsman. "Distributed Computation Meets Design Theory: Local Scheduling for Disconnected Cooperation." *Current Trends in Theoretical Computer Science: The Challenge of the New Century*, vol. 1: *Algorithms and Complexity*, pp. 315–336, World Scientific, 2004.

†Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA. Email: acr@cse.uconn.edu. The work of the first author is supported in part by the NSF Career Award 0093065, NSF ITR Grant 0220264, and the NSF Theory of Computing Grant 0311368,

‡Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269, USA and Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge MA, 02139, USA. Email: aas@cse.uconn.edu. The work of the second author is supported in part by the NSF Career Award 9984774, NSF Theory of Computing Grants 9988304 and 0311368, and NSF ITR Grant 0121277.

1 Introduction

Computation and communication are two basic resource commodities in distributed computing. In a distributed system setting one normally computes to perform specific tasks. One also needs to communicate in order to coordinate multiple computation activities, for example, to increase computation efficiency by eliminating redundant work. As can be expected, there exists a trade-off between the efficiency of distributed computation and the amount of communication needed to coordinate the computation. Dwork, Halpern and Waarts [8] have considered a basic problem of distributed coordination, where a number of message-passing processors cooperate in executing a collection of independent and similarly-sized tasks (in the presence of processor crashes). They measure the efficiency of the computation in terms of its *work* complexity W (one task consumes one work unit), and the cost of communication in terms of the *message* complexity M. The authors also assess the overall performance in terms of *effort* that is defined as the sum of work W and message complexity M. Chlebus *et al.* [4] solve a similar problem and they measure the effort of their solution as the sum of the number of all processing steps performed (this includes task-oriented work and any bookkeeping steps) and the number of messages sent. They observe that a processor can learn that a task is performed either by doing the task, or by receiving a message that tells it that the task was done. In such a setting, one can view computation and communication as comparable resources.

To study aspects of the trade-off between communication and computation in distributed cooperative applications, we consider the following abstract problem: *n processors must perform t tasks and learn the results of all tasks.* We assume that all tasks are known to all processors. A common impediment to effective coordination in distributed settings is asynchrony that manifests itself, for example, in disparate processor speeds and nondeterministic message latency. Fortunately, our problem can always be solved by a communication-oblivious algorithm that forces each processor to perform all tasks. Such a solution has work $W = O(t \cdot n)$, an requires no communication, i.e., $M = 0$. On the other hand, $\Omega(t)$ is the obvious lower bound on work and the best known lower bound is $W = \Omega(t + n \log n)$, cf. [16]. Therefore the trade-off expectation is that if we gradually increase the number of messages we should be able to decrease the amount of work performed.

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 t tasks. Consider the computation with a single rendezvous. There are $n-2$ processors that are unable to communicate, and they collectively must perform exactly $t \cdot (n-2)$ 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 $t/2$ 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, the could schedule their work as follows: one processors performs the tasks in the order $1, 2, \ldots, t$, the other in the order $t, t-1, \ldots, 1$. No matter when they happen 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 t 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 [7]) 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 t 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.

Related work: Cooperation with limited communication. The efficiency of work-performing algorithms depends on how well the loads are balanced among the participating processors and on the ability of the processors to disseminate information on the progress of the computation.

Papadimitriou and Yannakakis [22] 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 $t \geq 1$.

Georgiades, Mavronicolas, and Spirakis [10] 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 a variation of the problem we deal with in this report, Dolev *et al.* [6] 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 [6] develops an effective strategy for managing the task execution redundancy and proves that the strategy provides each of the n processors with a schedule of $\Theta(n^{1/3})$ tasks such that at most one task is performed redundantly by any two processors.

Structure of this report. In Section 2 we introduce our schedules, the "waste" measure and design. In Section 3 we present and prove the main lower bound on waste of schedules. Section 4 contains material showing that random schedules have good waste properties and that they furthermore behave competitively for arbitrary patterns of processor rendezvous. In Section 5 we present design-theoretic constructions that yield deterministic schedules with good waste properties. We conclude in Section 6.

2 Schedules, waste, and designs

We consider the abstract setting where n processors need to perform t independent and idempotent tasks. A task is *idempotent* if the execution of the task yields the same result when it is performed more than once. A task is independent if the result of its execution does not depend on the order in which other tasks are executed. The processors have unique identifiers from the set $[n] = \{1, \ldots, n\}$, and the tasks have unique identifiers from the set $[t] = \{1, \ldots, t\}$. Initially each processor knows the tasks that need to be performed and their identifiers.

We shall focus on the scheduling problem discussed above, abstracted as follows. An (n,t)-*schedule* is a tuple $(\sigma_1, \ldots, \sigma_n)$ of n permutations of the set $[t]$. When $n = 1$ it is elided and we simply write *t-schedule*.

An (n,t)-schedule immediately gives rise to a strategy for n isolated processors who must complete t 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 some k of these processors, say $q_1, \ldots, q_k$, should rendezvous at a time when the ith processor in this group, q_i, has completed a_i tasks. 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*:

Definition 2.1. *If S is a (n,t)-schedule and $(a_1, \ldots, a_k) \in \mathbb{N}^k$, the* waste *function for S is*

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

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

Here (and throughout), if $\phi : X \to Y$ is a function and $S \subset X$, we let $\phi(S) = \{\phi(x) \mid x \in S\}$. For a specific vector $a = (a_1, \ldots, a_k)$, $\mathcal{W}_S(a)$ 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 $\mathcal{W}_S(\cdot,\cdot)$ can be naturally extended to bounds on *k-wise* waste $\mathcal{W}_S(\underbrace{\cdot,\ldots,\cdot}_{k})$: specifically, note that if S is an (n,t)-schedule then

$$\mathcal{W}_S(a_1, \ldots, a_k) \le \sum_{i<j} \mathcal{W}_S(a_i, a_j)$$

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 4.2. With this justification we shall content ourselves to focus the investigation on pairwise waste—the function $\mathcal{W}_S(a,b)$.

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 (see, e.g., [14] for a survey). Despite this, the study of *waste* appears to be new. We shall, however, make substantial use of some design-theoretic constructions, which we describe below.

Definition 2.2. *A ℓ-(v,k,λ)* design *is a family of subsets $\mathcal{S} = (S_1, \ldots, S_n)$ of the set $[v]$ with the property that each $|S_i| = k$ and any set of ℓ elements of $[v]$ is a subset of precisely λ of the S_i. (N.B. The subsets S_i are typically referred to as* blocks.*)*

Observe that if $\mathcal{S}$ 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 S_j. Of course, if U_i is a subset of some S_j, then in fact exactly $k - (\ell - 1)$ if the U_i are subsets of S_j. Hence T appears in exactly $\lambda(v - \ell + 1)/(k - \ell + 1)$ of the S_j, as desired.

To see the connection between such designs and our problem, let $\mathcal{D}$ be a 2-(n,k,λ) design consisting of t sets $S_1, \ldots, S_t$. For each $i \in [n]$, let $T_i = \{j \mid i \in S_j\}$. Note now that for any $i \ne j$,

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

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 $\Sigma = (\sigma_1, \ldots, \sigma_n)$ be any sequence of permutations of $[t]$ for which $\sigma_i([a]) = T_i$. It is clear that these form an (n, t)-schedule for which

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

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 $\sigma_i([n] \setminus [a])$, 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 $n = q^2 + q + 1$ distinct one dimensional subspaces of $\mathrm{GF}(q)^3$, which we denote $\ell_1, \ldots, \ell_n$. We say that two subspaces ℓ_i and ℓ_j are *orthogonal* if $\forall u \in \ell_1, \forall v \in \ell_2$, $\langle u, v\rangle = \sum u_j v_j \bmod p = 0$; 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. See [14] for a proof and more discussion.

For concreteness, we fix a specific (arbitrary) ordering of each of these sets S_u: let L_u denote a canonical sequence $\langle t_u^1, \ldots, t_u^r\rangle$ where $S_u = \{\ell_{t_u^i} \mid 1 \leq i \leq q+1\}$; i.e., the one dimensional subspaces $\ell_{t_u^i}$, $i = 1, \ldots, q+1$, 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 theorem.

Theorem 2.1. *Let $n = q^2 + q + 1$, where q is a prime power. Then the sequences $\mathcal{L}_n = \langle L_1, \ldots, L_n\rangle$ possess the following properties: each L_u has length $q + 1$, for each $u \neq v$, $|L_u \cap L_v| = 1$, 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 $O(\log n)$ time; each subsequent element can be calculated in $O(1)$ time.*

(We assume throughout that addition or multiplication of two $\log(\max\{n, t\})$-bit numbers can be performed in $O(1)$ time.)

3 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 $\Omega(t \cdot g)$, even if each group of processors performs no more than the optimal number of tasks, t. In the extreme case where all processors are isolated from the beginning, the work of any algorithm is $\Omega(t \cdot n)$, 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 $\Omega(t \cdot n)$ 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 $t/2$ 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 n total processors, any two processors, having executed the same number of tasks t', where $t' < t$, perform *no* redundant work, then it must be the case that $t' \leq \lfloor t/n \rfloor$. In particular, if $n = t$, 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 t_1 and t_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 t_1 and t_2, then grows quadratically in the schedule length as t_1 and t_2 approach t.

We begin with a short geometric lemma.

Lemma 3.1. *Let $\mathbf{a}, \mathbf{b} \in \mathbb{R}^t$ and suppose that $\forall i, 0 \leq a_i \leq b_i \leq M$ and that $\|\mathbf{b}\|_1 - \|\mathbf{a}\|_1 = \kappa M$ for some positive $\kappa \in \mathbb{N}$. Then*

$$\langle \mathbf{a}, \mathbf{b} \rangle \geq \frac{\|\mathbf{a}\|_1^2}{t - \kappa}.$$

Proof. We say that a vector $\mathbf{x} \in \mathbb{R}^t$ is *non-negative* if $\forall i, x_i \geq 0$; for two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^t$, we write $\mathbf{x} \leq \mathbf{y}$ if $\mathbf{y} - \mathbf{x}$ is non-negative. Let $\mathbf{e_i}$ denote the standard basis vectors for $\mathbb{R}^n$. For the non-negative vector $\mathbf{b} \in \mathbb{R}^t$, consider the following family of transformations: for each distinct $r, s \in [t]$ and non-negative $\mathbf{x} \leq \mathbf{b}$, define

$$T_{\mathbf{b}}^{rs}(\mathbf{x}) = \mathbf{x} + \Delta \cdot (\mathbf{e_r} - \mathbf{e_s})$$

where $\Delta = \min(b_r - x_r, x_s)$. Note that $\|T_{\mathbf{b}}^{rs}(\mathbf{x})\|_1 = \|\mathbf{x}\|_1$ and $T_{\mathbf{b}}^{rs}(\mathbf{x}) \leq \mathbf{b}$. Evidently, if $\mathbf{x}$ and $\mathbf{b}$ satisfy the conditions of the theorem above, then so do $T_{\mathbf{b}}^{rs}(\mathbf{x})$ and $\mathbf{b}$, with the same κ. Note further that if $b_r \leq b_s$ then

$$\langle T_{\mathbf{b}}^{rs}(\mathbf{x}), \mathbf{b} \rangle = \langle \mathbf{x}, \mathbf{b} \rangle + \Delta \cdot (b_r - b_s) \leq \langle \mathbf{x}, \mathbf{b} \rangle.$$

Let $Z \subset [n]$ denote a set of κ indices so that $b_i \geq b_j$ for all $i \in Z$ and $j \notin Z$. Now, if $a_z = 0$ for all $z \in Z$ then $\mathbf{a}$ is supported on the set $[n] \setminus Z$ of size $t - \kappa$ so that

$$\langle \mathbf{b}, \mathbf{a} \rangle \geq \langle \mathbf{a}, \mathbf{a} \rangle \geq \frac{\|\mathbf{a}\|_1^2}{t - \kappa},$$

by the Cauchy-Schwarz inequality. Otherwise there is an index $z \in Z$ so that $a_z > 0$, and in this case there must be an index $y \notin Z$ so that $a_y < b_y$ since $\|\mathbf{b} - \mathbf{a}\|_1 = \kappa M$ but $\sum_{i \in Z}(b_i - a_i) \leq \kappa M - a_z < \kappa M$. Let $\mathbf{a}' = \mathbf{a}^{(1)} = T_{\mathbf{b}}^{yz}(\mathbf{a})$, and observe that $\sum_{i \in Z} a_i' < \sum_{i \in Z} a_i$. A finite number of iterations of this process results in a vector $\mathbf{a}^{(k)} \leq \mathbf{b}$ for which $\mathbf{a}_i^{(k)} = 0$ for all $i \in Z$, $\|\mathbf{a}\|_1 = \|\mathbf{a}^{(k)}\|_1$ and $\langle \mathbf{b}, \mathbf{a} \rangle \geq \langle \mathbf{b}, \mathbf{a}^{(k)} \rangle$. Then, by the same reasoning as above,

$$\langle \mathbf{b}, \mathbf{a} \rangle \geq \langle \mathbf{b}, \mathbf{a}^{(k)} \rangle \geq \langle \mathbf{a}^{(k)}, \mathbf{a}^{(k)} \rangle \geq \frac{\|\mathbf{a}^{(k)}\|_1^2}{t - \kappa} = \frac{\|\mathbf{a}\|_1^2}{t - \kappa}.$$

□

Now we proceed to the lower bound, which generalizes the second Johnson Bound [15] for the case when two processors execute *different* number of tasks prior to their rendezvous.

Theorem 3.2 ([18]). *Let $\mathcal{P} = \langle \pi_1, \ldots, \pi_n \rangle$ be an (n, t)-schedule and let $0 \leq a \leq b \leq t$. Then*

$$\mathcal{W}_{\mathcal{P}}(a, b) \geq \frac{na^2}{(n-1)(t-b+a)} - \frac{a}{n-1}.$$

Proof. We obtain the lower bound by computing the expected waste of a pair of t-schedules selected at random from $\mathcal{P}$. Let $\lambda = \mathcal{W}_{\mathcal{P}}(a, b)$. Consider selection of i and j independently at random in the set $[n]$. We focus on the expected value of the random variable

$$|\pi_i([a]) \cap \pi_j([b])|.$$

There are a total of n^2 pairs for i and j; if $i \neq j$ then the cardinality of the intersection is bounded above by λ. If $i = j$ then this cardinality is obviously a. Hence

$$\mathbf{E}[|\pi_i([a]) \cap \pi_j([b])|] \leq \frac{n(n-1)\lambda + n \cdot a}{n^2} = \frac{\lambda(n-1) + a}{n}. \tag{1}$$

Consider now the t random variables X_τ, indexed by $\tau \in [t]$, defined as follows: $X_\tau = 1$ if $\tau \in \pi_i([a]) \cap \pi_j([b])$, and 0 otherwise. Then $\mathbf{E}[|\pi_i([a]) \cap \pi_j([b])|] = \mathbf{E}[\sum_{\tau \in [t]} X_\tau]$ and by linearity of expectation,

$$\mathbf{E}[|\pi_i([a]) \cap \pi_j([b])|] = \sum_{\tau \in [t]} \mathbf{E}[X_\tau] = \sum_{\tau \in [t]} \Pr[\tau \in \pi_i([a])] \cdot \Pr[\tau \in \pi_j([b])],$$

since i and j are independent.

Now we introduce the function $x^m(\tau)$, equal to the number of prefixes of schedules of length m to which τ belongs, i.e., $x^m(\tau) = |\{i : \tau \in \pi_i([m])\}|$. Then

$$\mathbf{E}[|\pi_i([a]) \cap \pi_j([b])|] = \sum_{\tau \in [t]} \Pr[\tau \in \pi_i([a])] \cdot \Pr[\tau \in \pi_j([b])] = \sum_{\tau \in [t]} \frac{x^a(\tau) x^b(\tau)}{n^2} = \frac{1}{n^2} \sum_{\tau \in [t]} x^a(\tau) x^b(\tau) . \quad (2)$$

Noting that $\sum x^a(\tau) = an$ and $\sum x^b(\tau) = bn$, we apply Lemma 3.1 to the last expression in (2) above and combine this with the bound of (1):

$$\frac{1}{n^2} \cdot \frac{(na)^2}{(t-b+a)} \leq \frac{1}{n^2} \sum_{\tau \in [t]} x^a(\tau) x^b(\tau) \leq \mathbf{E}[|\pi_i([a]) \cap \pi_j([b])|] \leq \frac{(n-1)\lambda + a}{n}$$

whence

$$\lambda \geq \frac{a}{n-1} \left(\frac{na}{t-b+a} - 1 \right),$$

as desired. □

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

Corollary 3.3 ([18]). *For $t = n$, if $a > \sqrt{n - 3/4} + \frac{1}{2}$ then any n-processor schedule of length a for t tasks has worst case pairwise waste at least 2.*

4 Random schedules

As one would expect, schedules chosen at random perform quite well. In this section we explore the behavior of the (n,t)-schedules obtained when each permutation is selected uniformly (and independently) at random among all permutations of $[t]$.

4.1 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 $\pi_i \in S_{[t]}$ which determines the order in which this processor will complete the tasks. ($S_{[t]}$ denotes the collection of all permutations of the set $[t]$.) 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/t. 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 4.1 ([18]). *Let $\mathcal{R}$ be a system of n random schedules for t tasks constructed as above. Then with probability at least $1 - \frac{1}{nt}$,*

$$\forall a, b \text{ such that } 7\sqrt{t}\ln(2nt) \leq a, b \leq t, \quad \mathcal{W}_{\mathcal{R}}(a,b) \leq \frac{ab}{t} + \Delta(a,b) \ ,$$

where $\Delta(a,b) = 11\sqrt{\frac{ab}{t}\ln(2nt)}$

Observe that Theorem 3.2 shows that (n,t)-schedules must have waste $\mathcal{W}(a,a) = \Omega(a^2/t)$ (as $n \to \infty$); hence such randomized schedules offer nearly optimal waste for this case.

4.2 k-Waste for random schedules

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

Theorem 4.2. *Consider the random schedule $\mathcal{R}$ from above. Then with probability at least $1 - 1/n$,*

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

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

Note that again this bounds the distance of the k-waste from its expected value, which can be computed by inclusion-exclusion to be $\sum_{s=2}^{k}(-1)^s\binom{k}{s}\frac{a^s}{t^{s-1}}$. 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. (See [1] for a discussion of discrete exposure martingales and Azuma's inequality.)

4.3 Arbitrary rendezvous patterns

Thus far we have established bounds on wasted work for a single rendezvous. It is naturally interesting to study the general case allowing arbitrary patterns of (not necessarily pairwise) rendezvous. Of course, an adversary that partitions the processors into g disconnected components during the entire the life of the computation causes any task-performing algorithm to have work $\Omega(t \cdot g)$, even if each group of processors performs no more than the optimal number of $\Theta(t)$ tasks. This lower bound would appear to form a somewhat pessimistic landscape for our problem. Considering, however, that *no* algorithm can maintain low total work in the presence of such pathological communication failures, it seems reasonable to pursue a *competitive* analysis [21] for this general framework, and compare the behavior of a given algorithm with that of an optimal algorithm.

In particular, we consider the *partitionable network* scenario consisting of n asynchronous processors with a communication medium that is subject to arbitrary *partitions* during the life of the computation. This model is motivated by the abstraction provided by a typical *group communication scheme*; see, for example, [3] and the surveys in [5]. Specifically, at each point of the computation, we assume that the communication medium effectively partitions the processors into non-overlapping *groups*: communication within a group is instantaneous and reliable, communication across groups is impossible. Naturally, processors in the same group can share their knowledge of completed tasks and, while they remain connected, avoid doing redundant work. We refer to a transition from one partition to another as a *reconfiguration*.

Our goal is to design schedules that minimize the total *work*, where work is defined to be *the number of tasks executed by all the processors during the entire computation (counting multiplicities)*. Ideally, when two processors "meet" in a new group (during a reconfiguration) the sets of tasks they know to be complete would be disjoint to avoid wasted effort. This is impossible in general, as processors must schedule their

work in ignorance of future reconfigurations and, moreover, circumstances where two processors meet who have collectively completed more than t tasks will necessitate wasted work. (It is, of course, also possible that the two processors were members of a common group during a previous portion of the computation, resulting in shared knowledge.) A processor may cease executing tasks *only* when it knows the results of all tasks. We refer to this problem as *Omni-Do*.

We do not charge for coordination within a group, simply treating grouped processors as a single (virtual) asynchronous processor. In particular, if a group of processors performs a set of t tasks during the lifetime of the group, we charge this group t units of work, ignoring, for example, partially completed tasks which may remain at the group's demise or the cost of synchronizing processors' knowledge during the group's inception. Note that while processors are asynchronous, they do not crash.

An algorithm in this model is a rule which, given a group of processors and a set of tasks known by this group to be complete, determines a task for the group to complete next. In the case where all processors are disconnected during the entire computation, any algorithm must incur $\Theta(t \cdot n)$ work. On the other hand, any reasonable algorithm should attain t work in the case where all processors remain connected during the computation.

We consider the behavior of an algorithm in the face of an adversary (which is *oblivious* in the sense of [2]) that determines both the *sequence of reconfigurations* and the *number of tasks completed* by each group before it is involved in another reconfiguration. Taken together, this information determines a *computation pattern*: this is a directed acyclic graph (DAG), each vertex of which corresponds to a group G of processors that existed during the computation; a directed edge is placed from G_1 to G_2 if G_2 was created by a reconfiguration involving G_1 and the two groups have at least one processor in common. 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. Note that different adversaries (causing different sequences of reconfigurations) may give rise to the same computation pattern; the *work* caused by an adversary, however, depends only on the computation pattern determined by that adversary.

Specifically, if t is the number of tasks and n the number of processors, then such a computation pattern is a labeled and weighted directed acyclic graph, that we call a (n,t)-DAG:

Definition 4.1. *A (n,t)-DAG is a directed acyclic graph $C = (V,E)$ augmented with a weight function $h : V \to \mathbb{N}$ and a labeling $g : V \to 2^{[n]} \setminus \{\emptyset\}$ so that:*

(i) For any maximal path $\mathbf{p} = (v_1, \ldots, v_k)$ in C, $\sum h(v_i) \geq t$. (This guarantees that any algorithm terminates during the computation described by the DAG.)

(ii) g possesses the following "initial conditions":

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

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

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

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

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

Here $\dot{\cup}$ denotes disjoint union and $in(v)$ and $out(v)$ denote the in-degree and out-degree of v, respectively.

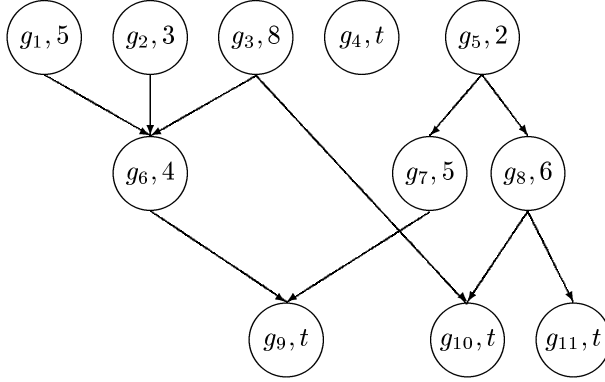

Figure 1: An example of a $(12, t)$-DAG

EXAMPLE. A sample $(12, t)$-DAG is shown in Figure 1. Here we have $g_1 = \{p_1\}$, $g_2 = \{p_2, p_3, p_4\}$, $g_3 = \{p_5, p_6\}$, $g_4 = \{p_7\}$, $g_5 = \{p_8, p_9, p_{10}, p_{11}, p_{12}\}$, $g_6 = \{p_1, p_2, p_3, p_4, p_6\}$, $g_7 = \{p_8, p_{10}\}$, $g_8 = \{p_9, p_{11}, p_{12}\}$, $g_9 = \{p_1, p_2, p_3, p_4, p_6, p_8, p_{10}\}$, $g_{10} = \{p_5, p_{11}\}$, and $g_{11} = \{p_9, p_{12}\}$.

This computation pattern models all asynchronous computations (adversaries) 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. □

We say that two groups G and G' are *independent* if there is no directed path connecting one to the other. For a computation pattern C, the *computation width* of C, denoted $\mathbf{cw}(C)$, is the maximum number of independent groups reachable (along directed paths) in this DAG from any vertex.

We consider a competitive analysis that compares the work of a randomized algorithm with the work of an optimal algorithm that has complete information about the computation history (and hence the future pattern of reconfigurations).

Let D be a deterministic algorithm for *Omni-Do* and C a computation pattern, we let $W_D(C)$ denote the total work expended by algorithm D, where reconfigurations are determined according to the computation pattern C. Work is formally defined as follows:

Definition 4.2. *Let C be a (n, t)-DAG and D a deterministic algorithm for Omni-Do. $W_D(C)$ is defined inductively as follows. For a vertex v of C with $in(v) = 0$, define L_v to be the set containing the first $h(v)$ tasks completed by group $g(v)$ according to D. Otherwise, $in(v) > 0$; in this case, let $\check{L}_v = \bigcup_{u<v} L_u$ denote the collection of all tasks known to be complete at the inception of the group $g(v)$. Then let L_v be the first $h(v)$ tasks completed by group $g(v)$ according to D starting with knowledge $\check{L}_v$. If $h(v) > t - |\check{L}_v|$, define $L_v = [t] \setminus \check{L}_v$. Then $W_D(C) = \sum_{v \in C} |L_v|$.*

We treat randomized algorithms as distributions over deterministic algorithms; for a set Ξ and a family of deterministic algorithms $\{D_r \mid r \in \Xi\}$ we let $R = \Re(\{D_r \mid r \in \Xi\})$ denote the randomized algorithm where r is selected uniformly at random from Ξ and scheduling is done according to D_r. For a real-valued random variable X, we let $\mathbf{E}[X]$ denote its expected value. We let OPT denote the optimal off-line algorithm, which may schedule tasks with full knowledge of the pattern of reconfigurations. Specifically, for each C we define $W_{\text{OPT}}(C) = \min_D W_D(C)$.

Definition 4.3 ([21, 9, 2]). *Let α be a real valued function defined on the set of all (n,t)-DAGs (for all n and t). A randomized algorithm R is α-**competitive** if for all computation patterns C,*

$$\mathbf{E}_r[W_{D_r}(C)] \leq \alpha(C) W_{\text{OPT}}(C),$$

this expectation being taken over uniform choice of $r \in \Xi$. The definition specializes naturally to the case of a deterministic algorithm.

We begin with a lower bound for *deterministic* algorithms. This is then applied to give a lower bound for randomized algorithms in Corollary 4.4.

Theorem 4.3 ([11]). *Let $a : \mathbb{N} \to \mathbb{R}$ and D be a deterministic scheduling 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 $a(c) \geq 1 + c/e$.*

This theorem is proved by considering a distribution on computation patterns C that is independent of the deterministic algorithm D—this immediately gives rise to a lower bound for randomized algorithms:

Corollary 4.4 ([11]). *Let $\Re(\{D_r \mid r \in \Xi\})$ be a randomized scheduling algorithm for the Omni-Do problem that is $(a \circ \mathbf{cw})$-competitive. Then $a(c) \geq 1 + c/e$.*

Unless the above lower bound is "too weak", it suggests that it is worthwhile to seek algorithms that are very competitive, despite the potentially high bounds on "absolute" work.

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

$$\text{RS} = \Re(\{D_\Pi \mid \Pi \in (S_{[t]})^n\}),$$

where $S_{[t]}$ is the collection of permutations on $[t]$. Let G be a group of processors and $\gamma \in G$ 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 $K \subset [t]$ have been completed, next completes the first task in the sequence $\pi_\gamma(1), \ldots, \pi_\gamma(t)$ which is not in K.

It turns out that this algorithm is optimal with respect to the lower bound on competitive ratios.

Theorem 4.5 ([11]). *Algorithm RS is $(1 + \mathbf{cw}(C)/e)$-competitive for any computation pattern C.*

5 Derandomization via finite geometries

We now consider a method for derandomizing these schedules using the design discussed in Section 2.

5.1 Schedules for $n = t$

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

Lemma 5.1. *Let $\mathcal{L}_n = \langle L_1, \ldots, L_n \rangle$ be the collection of sequences constructed in Theorem 2.1, and let $L_u = \langle t_u^1, \ldots, t_u^{q+1} \rangle$, $1 \le u \le n$. Then for any $i \neq j$, we have $L_{t_u^i} \cap L_{t_u^j} = \{u\}$.*

Proof. Consider any two distinct sequences $L_{t_u^i}$ and $L_{t_u^j}$, where $i \neq j$. By construction these sequences contain the indices corresponding to the one dimensional subspaces that are orthogonal to lines $\ell_{t_u^i}$ and $\ell_{t_u^j}$, respectively. Since t_u^i is appears in L_u, line $\ell_{t_u^i}$ is orthogonal to line ℓ_u. By the same argument line t_u^j is orthogonal to line ℓ_u. Hence u appears in both $L_{t_u^i}$ and $L_{t_u^j}$, but $|L_{t_u^i} \cap L_{t_u^j}| = 1$ by Theorem 2.1 so that the intersection cannot contain any other element. ☐

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

$$P_u = \langle u \rangle \circ (\bigcirc_{i \in L_u} (L_i - u)),$$

where $\circ$ denotes concatenation and $L_i - u$ denotes the sequence L_i with u deleted. Note now that since the total length of P_u is evidently $(q+1)q+1 = n$, each element of $[n]$ must appear exactly once in each P_u; these P_u thus rive rise to a family of permutations π_u, where $\pi_u(k)$ is the k element of P_u. Let $\mathcal{P} = (\pi_1, \ldots, \pi_n)$.

We conceptually divide the sequences P_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 5.2. *Let q be a prime power, $n = q^2 + q + 1$. Let $a = 1 + iq$, $b = 1 + jq$, $0 \le i, j \le q+1$. Then*

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

Proof. Consider any two t-schedules π_u and π_v of $\mathcal{P}_n$; let P_u and P_v be the corresponding sequences. As the first elements in these schedules are distinct, the intersection for $i = j = 0$ is zero. The case when i or j is zero is easy. Assume now that $i, j \ge 1$. Consider the $i \cdot j$ pairs of segments (I, J), where I (or J) is one of the first i (or j) segments of P_u (or P_v). The recursive construction guarantees that only one pair may have segments where $I \subseteq J$ or $J \subseteq I$. For this pair the overlap is at most $q+1$ because these may be the first segments in the schedules. For the remaining $ij - 1$ pairs the overlap is at most 1. The result follows. ☐

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 GF(q). When q is prime this can be done in $O(\log n)$ using the extended Euclidean algorithm. Other elements of the schedule can be found in $O(1)$ time.

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

5.2 Controlling waste for short prefixes

One disadvantage of $\mathcal{P}_n$ 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 P_u so that the first segment is distinct across the resulting schedules. This can be accomplished by finding a permutation $\rho : [n] \to [n]$ such that the sequence L_u contains task $\rho(u)$. (In other words ℓ_u must be orthogonal to $\ell_{\rho(u)}$.) This permutation can then be used to select distinct segments as the first segments of schedules in $\mathcal{P}_n$.

Consider the bipartite graph $G_n = (U_n, V_n, E_n)$ where $U_n = V_n = [n]$ and $n = q^2 + q + 1$; here q is a prime power. Both U_n and V_n can be placed in one-to-one correspondence with the one dimensional subspaces of $\mathrm{GF}(q)^3$. An edge is placed between $\ell_u \in U_n$ and $\ell_v \in V_n$ when they are orthogonal. Based on the structure of $\mathrm{GF}(q)^3$, it is not hard to show that G_n is $(q+1)$-regular. By Hall's theorem (see, e.g., [12]), 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 [13] that runs in time $O(\sqrt{|U| + |V|} \cdot |E|) = O(n^2)$.

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

$$G_u = \langle u \rangle \circ (L_{\rho(u)} - \{u\}) \circ (\bigcirc_{i \in L_u - \rho(u)} (L_i - u)).$$

Then γ_u is the permutation associated with G_u.

Theorem 5.3. *Let q be a prime power, $n = q^2 + q + 1$. Let $a = 1 + iq$, $b = 1 + jq$, $0 \leq i, j \leq q + 1$. Then:*

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

Proof. When $i = j = 1$ observe that by the construction of $\mathcal{G}_n$ the first segments of the schedules are distinct. The other cases follow the proof of Theorem 5.2. □

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

6 Conclusions

We surveyed results that characterize the ability of n isolated processors to collaborate on a common known set of t tasks. The good news is that the isolated processors can deterministically construct schedules locally, equipped only with the knowledge of n, t, and their unique processor identifiers in $[n]$. Moreover, the cost of constructing such schedules can be amortized over the performance of tasks. 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. We also show that when processors start working in isolation and are subjected to an arbitrary pattern of merges, randomized scheduling is competitive compared to an optimal algorithm that is aware of the pattern of merges.

Acknowledgements. Several results surveyed in this report were developed in collaboration with Greg Malewicz and Chryssis Georgiou. Additional related results and details are presented in Malewicz's doctoral dissertation [17].

References

[1] Alon, N. and Spencer, J.-H.: *The probabilistic method*. John Wiley & Sons Inc., New York, 1992. With an appendix by Paul Erdős, A Wiley-Interscience Publication.

[2] Ben-David, S, Borodin, A., Karp, R., Tardos, G. and Wigderson, A.: On the power of randomization in on-line algorithms. *Algorithmica*, 11(1):2–14, 1994.

[3] Birman, K.: The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, 1993.

[4] Chlebus, B., Gąsieniec, L., Kowalski, D., Shvartsman, A.A.: Bounding work *and* communication in robust cooperative computation. Proc. of 16th Int. Symposium on Distributed Computing, (2002) Springer LNCS 2508, 295–310

[5] *Comm. of the ACM*, Special Issue on Group Communication Services, vol. 39, no. 4, 1996.

[6] Dolev, S., Segala, R., Shvartsman, A.: Dynamic Load Balancing with Group Communication. *6th International Colloquium on Structural Information and Communication Complexity* (1999) 111-125 (to appear in *Theoretical Computer Science*).

[7] Dolev, D. and Malki, D.: "The Transis Approach to High Availability Cluster Communications", *Comm. of the ACM*, vol. 39, no. 4, pp. 64–70, 1996.

[8] Dwork, C., Halpern, J., Waarts, O.: Performing Work Efficiently in the Presence of Faults. *SIAM J. on Computing*, Vol. 27 **5** (1998) 1457–1491.

[9] Fiat, A., Karp, R.M., Luby, M., McGeoch, L.A., Sleator, D.D., and Young N.E.: Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[10] Georgiades, S., Mavronicolas, M., Spirakis, P.: Optimal, Distributed Decision-Making: The Case of No Communication. *Intl Symposium on Fundamentals of Computation Theory.* (1999) 293–303

[11] Georgiou, Ch., Russell, A., and Shvartsman, A.A.: Work-competitive scheduling for cooperative computing with dynamic groups. In *Proc. of the* 35th *ACM Symposium on Theory of Computing (STOC 2003)*, to appear, 2003. (Prelim. results reported in the brief paper: Optimally work-competitive scheduling for cooperative computing with merging groups. In *Proc. of the* 21st *ACM Symp. on Principles of Distributed Computing (PODC 2002)*, 2002.)

[12] Harary, F.: Graph Theory. Reading, MA: Addison-Wesley (1994)

[13] Hopcroft, J.E., Karp., R.M.: A $O(n^{5/2})$ algorithm for maximum matching in bipartite graphs. SIAM Journal on Computing, Vol. 2. (1973) 225–231

[14] Hughes, D.R. and Piper, F.C.: Design Theory, Cambridge University Press, 1985.

[15] Johnson, S.M.: A New Upper Bound for Error-Correcting Codes. IEEE Transactions on Information Theory, Vol. 8 (1962) 203–207

[16] Kanellakis, P.C., Shvartsman, A.A.: *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers (1997) ISBN 0-7923-9922-6.

[17] Malewicz, G.: *Distributed Scheduling for Disconnected Cooperation* Doctoral Dissertation, Computer Science and Engineering, University of Connecticut (2003).

[18] Malewicz, G., Russell, A., Shvartsman, A.A.: Distributed Cooperation During the Absence of Communication. 14th International Conference on Distributed Computing, LNCS Vol. 1914 (2000) 119–133

[19] Malewicz, G., Russell, A., Shvartsman, A.A.: Optimal Scheduling for Disconnected Cooperation. 8th International Colloquium on Structural Information and Communication Complexity (2001) 259-274 (Brief announcement. ACM Symposium on Principles of Distributed Computing. (2001))

[20] Malewicz, G., Russell, A., Shvartsman, A.: Local Scheduling for Distributed Cooperation. Invited paper, IEEE International Symposium on Network Computing and Applications, NCA'01 (2001)

[21] Sleator, D. and Tarjan, R.: Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[22] Papadimitriou, C.H., Yannakakis, M.: On the value of information in distributed decision-making. *ACM Symposium on Principles of Distributed Computing.* (1991) 61–64