

# Distributed Cooperation and Adversity: Complexity Trade-Offs\*

Chryssis Georgiou<sup>1</sup>  
cg2@cse.uconn.edu

Alexander Russell<sup>1</sup>  
acr@cse.uconn.edu

Alex A. Shvartsman<sup>1,2</sup>  
aas@cse.uconn.edu

<sup>1</sup>Dept. of Computer Science & Engineering  
University of Connecticut  
Storrs, CT 06269

<sup>2</sup>Laboratory for Computer Science  
Massachusetts Institute of Technology  
Cambridge, MA 02139

## ABSTRACT

The problem of cooperatively performing a collection of tasks in a decentralized setting where the computing medium is subject to adversarial perturbations is one of the fundamental problems in distributed computing. Such perturbations can be caused by processor failures, unpredictable delays, and communication breakdowns. To develop efficient distributed solutions for computation problems ranging from distributed search such as SETI to parallel simulation and multi-agent collaboration, it is important to understand efficiency trade-offs characterizing the ability of  $p$  processors to cooperate on  $t$ -tasks in the presence of adversity. This paper surveys recent results grouped by the following topics: (i) failure-sensitive bounds for distributed cooperation problems for synchronous processors subject to crash failures, (ii) bounds on redundant work for distributed cooperation when individual asynchronous processors may experience prolonged absence of communication, and (iii) competitive analysis of cooperative work performed by groups of asynchronous processors, when the groups may be fragmented and merged during the computation. These research results are motivated by the earlier work of the third author with Paris C. Kanellakis at Brown University.

## Categories and Subject Descriptors

C.4 [Computer System Organization]: Performance of Systems—*fault tolerance*; F.1.2 [Computation by Abstract Devices]: Modes of Computation—*parallelism and concurrency*; F.2.3 [Analysis of Algorithms and Problem Complexity]: Tradeoffs between Complexity Measures

\*This work is supported in part by the NSF Grant 9988304 and NSF ITR Grant 0121277. The work of the second author is supported in part by the NSF CAREER Award 0093065. The work of the third author is supported in part by the NSF CAREER Award 9984778.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PCK 50, June 8, 2003, San Diego, California, USA.

Copyright 2003 ACM 1-58113-604-8/03/0006 ...\$5.00.

## General Terms

Algorithms, Reliability, Theory

## Keywords

Distributed algorithms, performing tasks, fault-tolerance, work complexity, competitive analysis, scheduling, partitionable networks.

## 1. INTRODUCTION

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, GRID computing, and multi-agent collaboration. The benefit of solving a problem using multiple processors can only be realized if one is able to effectively marshal the available computing resources in order to achieve substantial speed-up relative to the time necessary to solve the problem using a single processor. In order to achieve high speed-ups it is necessary to eliminate redundant computation done by the processors. This is challenging because the availability of distributed computing resources may fluctuate due to failures and delays. Thus a system containing unreliable and asynchronous resources must dedicate resources both to solving the computational problem and to coordinating the available resources.

There is an intuitive trade-off between dependability and efficiency because reliability requires *introducing redundancy* in the computation in order to detect failures and delays, and to reassign resources, whereas gaining efficiency by parallel computing requires *removing redundancy* from the computation to fully utilize each processor. Thus, even allowing for some abstraction in the model of parallel computation, it was not obvious that there are any non-trivial fault models that allow near-linear speed-ups. So it was somewhat surprising when Kanellakis and Shvartsman [29] demonstrated that it is possible to combine deterministic efficiency and fault tolerance for most basic algorithms expressed as concurrent-read concurrent-write (CRCW) parallel random access machines (PRAMs). Specifically, the failure model [29] allows *any pattern of dynamic processor crashes, as long as one processor remains alive*. Kanellakis and Shvartsman introduced a key abstract problem, called Write-All, and showed how to use solutions for this problem to construct dependable algorithms for other problems that already have efficient parallel algorithms that assume failure-

free and delay-free processors. The Write-All problem is formulated in terms of  $p$  processors writing to  $t$  distinct memory locations in the presence of an adaptive adversary that introduces dynamic failures or delays. Here writing to a memory location models an independent task that can be performed by a single processor in constant time. The efficiency of dependable shared-memory algorithms is measured according to *available processor steps* introduced in [29] (also referred to as *work*), now a common complexity measure in the study of fault-tolerant algorithms. Among the noteworthy early results is the best known Write-All algorithm for synchronous crash-prone processors, and the tight lower bound on work for processors with constant-time memory snapshots. A monograph by Kanellakis and Shvartsman [28] brought together several important results in this area.

Following the initial work [29], the Write-All problem was studied in a variety of shared-memory settings e.g., [2, 3, 6, 21, 27, 30, 31, 32, 35, 40, 41, 42]. Dwork, Halpern and Waarts [11] extended Write-All to message-passing models of computation, where the problem, called Do-All, is stated in terms of  $p$  processors that need to perform  $t$  independent and idempotent tasks in the presence of failures. Many algorithms were developed to solve Do-All in a variety of message-passing settings, e.g., [7, 8, 9, 11, 14, 33, 43]. The problem has also been studied in partitionable networks, e.g., [10, 19, 20, 36]. Depending on the specific model of computation, the efficiency of solutions for these problems is assessed in terms of work (task-oriented work, total work, or redundant work), time, and communication complexity.

In this work we survey the state-of-the-art results in the following three areas: (i) failure-sensitive bounds for distributed cooperation problems such as Write-All and Do-All (Section 2), based on [17, 18], (ii) bounds on redundant work for distributed cooperation during the prolonged absence of communication (Section 3), based on [36, 37, 38], and (iii) competitive analysis of cooperative work performed by fragmenting and merging groups of processors (Section 4), based on [19].

The results presented here are directly motivated by the research done by the third author, Alex Shvartsman, with Paris Kanellakis from 1988 to 1992 at Brown University, when Paris served as his doctoral advisor. It is appropriate to repeat here the following acknowledgment from Shvartsman’s dissertation [50]:

*I thank my advisor Paris C. Kanellakis who provided the initial vision for this work. His scientific intuition, well-founded optimism, research discipline, and contagious energy greatly contributed to the success of our work. Working together through long brainstorming sessions were the most memorable and enjoyable times during my stay at Brown.*

## 2. FAILURE-SENSITIVE BOUNDS

Performing a set of tasks in a decentralized setting is a fundamental problem in distributed computing. This is often challenging because the set of processors available to the computation and their ability to communicate may dynamically change due to perturbations in the computation medium. An abstract statement of this problem, referred to as the Do-All problem:

*$p$  fault-prone processors perform  $t$  independent tasks,*

is one of the standard problems in the research on the complexity of fault-tolerant distributed computation [11, 29].

Solutions for Do-All must perform all tasks efficiently in the presence of specific failure patterns. The efficiency is assessed in terms of work, time and communication complexity depending on the specific model of computation.

We consider abstract models of computation where  $p$  synchronous processors are subject to  $f$  crashes ( $f < p$ ). Here a processor crash is a stop-failure [48], such that the processor halts and performs no further actions. We measure the efficiency of solutions for Do-All (or Write-All) in terms of *work* complexity where all processing steps taken by all processors are counted. Work is ideally expressed as a function of  $p$ ,  $t$ , and  $f$ . Until very recently, an unsatisfactory landscape existed with respect to the understanding of how the bounds on work depend on  $f$ , the number of failures. That is, work was typically given as a function of  $t$  and  $p$ , but it was either not elucidated how  $f$  impacts work, or, when  $f$  was a part of the equation, it was primarily due to the nature of a specific algorithm, and not due to the inherent properties of the Do-All problem. For example, the work of the best known synchronous shared-memory algorithm is given as a function of  $p$  and  $t$  [29]. This is also the case with the best known asynchronous shared-memory algorithm [2]. Similarly, the best known lower bound for shared-memory models [31] and the best known lower bound applicable to message-passing models [6] do not involve  $f$ . The work of message-passing algorithms, e.g., [9, 14], typically *does* include  $f$ , but this is due to the use of single coordinators, which means that for  $f$  coordinator failures the work necessarily includes an additive term  $f \cdot p$ . A message-passing algorithm using multiple coordinators [7] avoids this inefficiency and includes a term that depends on  $\log f$ , but this involves  $f$  in a somewhat superficial way. Thus prior lower/upper bound results for Do-All do not teach adequately how the work complexity depends on the number of failures  $f$ .

When considering synchronous shared-memory computing with failure-prone processors the impact of imprecise analysis of work complexity is especially significant. Approaches such as [32, 42, 49] use the *iterative* Do-All approach to execute synchronous parallel (PRAM) algorithms on failure-prone processors by simulating the parallel steps of ideal processors with the help of some chosen Do-All algorithm. It was shown that the execution of a single  $t$ -processor step on  $p$  failure-prone processors does not exceed the complexity of solving a  $t$ -size instance of Do-All using  $p$  failure-prone processors. Thus if  $W_{t,p}$  is the complexity of solving a Do-All instance of size  $t$  using  $p$  processors, and the *parallel-time*  $\times$  *processor* product of the given synchronous  $t$ -processor,  $r$ -time algorithm is  $r \cdot t$ , then the algorithm can be deterministically simulated with work  $O(r \cdot W_{t,p})$ . If the analysis does not accurately reflect the impact of the number of failures  $f$ , then we shall see that the resulting upper bound is needlessly inflated. Even if we compute the upper bounds on work of the iterative use of Do-All as the product  $r \cdot W_{t,p,f}$ , where  $r$  is the number of iterations and  $W_{t,p,f}$  is the failure-sensitive upper bound for the Do-All problem, this result may still be inflated, since fewer than  $f$  failures may be “available” to the adversary per iteration when  $r > 1$ . Practical concerns would be well served by the knowledge of what happens in Do-All when the number of failures is moderate and when the Do-All algorithms are used iteratively. In particular, it is important to understand the behavior of the best algorithms for the entire range of failures  $f$  and iterations  $r$ .

## 2.1 Bounds for perfect load balancing

We let  $\text{Do-All}(t, p, f)$  stand for the  $\text{Do-All}$  problem for  $t$  tasks,  $p$  processors and up to  $f$  crashes. The processors have unique identifiers (PIDs) from the set  $[p] = \{1, \dots, p\}$ , and the tasks have unique identifiers from the set  $[t] = \{1, \dots, t\}$ . We let  $\text{Do-All}^{\mathcal{O}}(t, p, f)$  denote the  $\text{Do-All}(t, p, f)$  problem that is solved with the use of an oracle  $\mathcal{O}$  that may make available to the processors a deterministic function of the global history of the computation: in particular, the oracle may be used to assist the processors to load-balance and terminate (but unlike the oracle's Delphian colleague, it cannot predict the future). The oracle assumption is used as a *tool* for studying the work complexity patterns of *any* fault-tolerant algorithm: (i) of course, any lower bound developed for this strong model applies equally well to weaker specialized message-passing or shared memory models, (ii) an algorithm in the oracle model may be simulated in a message-passing or shared memory model by suitably approximating the oracle. As most  $\text{Do-All}$  algorithms use communication to load-balance and detect termination, this framework allows for the complexity analysis of specific algorithms to be divided into two parts: (1) the analysis of the cost of tolerating failures while performing work assuming perfect load-balancing, and (2) the analysis of the cost of implementing perfect load-balancing. We used this approach to derive new  $f$ -sensitive upper bounds for message-passing and shared-memory models.

One may study the complexity of  $\text{Do-All}^{\mathcal{O}}(\cdot)$  in conjunction with a variety of different oracles. When we consider lower bounds, we consider arbitrary oracles that may make available, for example, the entire global history of the computation on every processor. When we consider upper bounds, we use the *load-balancing* oracle,  $\mathcal{O}_L$ , described next. During each synchronous iteration of the computation, the oracle  $\mathcal{O}_L$  makes available to each processor  $P_i$  two values: *Oracle-complete*, a Boolean which takes the value true if and only if all tasks were completed at the beginning of this iteration, and *Oracle-task<sub>i</sub>*, an integer whose value is a task identifier with the property that the identifiers associated to the live processors are split evenly among the tasks. In particular, if processors  $i_1, \dots, i_k \in [p]$  are alive and tasks  $j_1, \dots, j_\ell \in [t]$  are incomplete at the beginning of the iteration, then *Oracle-task<sub>i<sub>m</sub></sub>* =  $j_n$ , where  $n = (m - 1 \bmod \ell) + 1$ .

With foresight, we define below a quantity that allows us to abbreviate the presentation of our results. We define the quantity  $\Lambda_{t,p,f}^r$ , where  $r$  is the number of  $\text{Do-All}$  iterations (for  $r$ -iterative  $\text{Do-All}$ ), and where  $t, p$ , and  $f$  appear in their usual roles, as follows.

$$\Lambda_{t,p,f}^r = \begin{cases} (a) \log\left(\frac{pr}{f}\right) & \text{when } f \leq \frac{pr}{\log(\min(t,p))} \\ (b) \log \log(\min(t,p)) & \text{when } f > \frac{pr}{\log(\min(t,p))} \end{cases}$$

For a single instance  $\text{Do-All}$ , i.e., when  $r = 1$ , we simplify this by defining  $\Lambda_{t,p,f} = \Lambda_{t,p,f}^1$ .

In presenting the upper bounds for  $\text{Do-All}^{\mathcal{O}}(t, p, f)$ , we consider the oracle-based algorithm in Figure 1. The oracle  $\mathcal{O}_L$  performs the termination and load-balancing computation on behalf of the processors.

The following upper bound can be shown by induction on the number of unperformed tasks.

**THEOREM 2.1** ([18]). *The  $\text{Do-All}^{\mathcal{O}_L}(t, p, f)$  problem can be solved for any pattern of crashes using work  $W(t, p, f) = \mathcal{O}(t + p + p \log(\min(t, p)) / \Lambda_{t,p,f})$ .*

```

for each processor PID = 1..p begin
  global T[1..t];
  while Oracle-complete() = false
    do perform task T[Oracle-task(PID)] od
end.
```

Figure 1: Oracle-based algorithm.

For the lower bounds we consider a specific adversarial strategy. Let  $A$  be an iterative algorithm that solves the  $\text{Do-All}^{\mathcal{O}}(t, p, f)$  problem. Let  $p_i$  be the number of processors remaining at the end of the  $i^{\text{th}}$  iteration of  $A$  and let  $u_i$  denote the number of tasks that remain to be done at the end of iteration  $i$ . Initially,  $p_0 = p$  and  $u_0 = t$ . The strategy of the adversary is defined for each iteration of the algorithm. Based on a variable  $\kappa$ , defined in the interval  $(0, 1/2)$ , the adversary determines which processors will be allowed to work and which will be crashed in a given iteration. We call this adversary  $\mathfrak{A}$ .

Adversary  $\mathfrak{A}$ :

*Iteration 1:* The adversary chooses  $u_1 = \lfloor \kappa u_0 \rfloor$  tasks with the least number of processors assigned to them. This can be done since the adversary is omniscient; it knows all the actions to be performed by  $A$  (as well as any advice provided by the oracle). The adversary then crashes the processors assigned to these tasks, if any.

*Iteration  $i$ :* Among  $u_{i-1}$  tasks remaining after the iteration  $i-1$ , the adversary chooses  $U_i = \lfloor \kappa u_{i-1} \rfloor$  tasks with the least number of processors assigned to them and crashes these processors.

*Termination:* The adversary continues for as long as  $u_i > 1$ . As soon as  $u_i = 1$ , the adversary allows all remaining processors to perform the single remaining task, and  $A$  terminates.

The next result is obtained by optimizing  $\kappa$ .

**THEOREM 2.2** ([18]). *For any oracle  $\mathcal{O}$  and for any algorithm that solves  $\text{Do-All}^{\mathcal{O}}(t, p, f)$ , there is an adversary that causes work  $W(t, p, f) = \Omega(t + p + p \log(\min(t, p)) / \Lambda_{t,p,f})$ .*

This lower bound applies to algorithms in more specialized models, e.g., message passing or shared memory models.

The bounds in Theorems 2.1 and 2.2 show, for example, that even with the help of the oracle, work grows rapidly as the number of crashes increases from 0 to  $p / \log p$ , at which point work reaches its maximum at  $\Theta(p \log p / \log \log p)$ , for  $p = t$ .

## 2.2 Upper bounds for message-passing and shared-memory models

We now show the utility of the complexity results under the perfect load-balancing assumption. We present bounds for two algorithms whose prior analyses did not integrate  $f$  adequately. This is done by analyzing how load-balancing is implemented by the algorithms, e.g., by using coordinators or global data-structures. The first algorithm, called AN [7], is an efficient synchronous message-passing algorithm for  $\text{Do-All}$  with  $f < p \leq t$ . The second algorithm, called W [29], is the best known synchronous shared-memory algorithm for  $\text{Write-All}$  with  $f < p \leq t$ .

## Message passing: Algorithm AN

Algorithm AN presented by Chlebus *et al.* [7] uses a multiple-coordinator approach to solve  $\text{Do-All}(t, p, f)$  on crash-prone synchronous message-passing processors. The model assumes that messages incur a known bounded delay and that reliable multicast [22] is available, however messages to/from faulty processors may be lost.

We give a brief description of the algorithm; additional details can be found in [7]. Algorithm AN proceeds in a *loop* which is iterated until all the tasks are executed. A single iteration of the loop is called a *phase*. A phase consists of three consecutive *stages*. Each stage consists of three steps. In each stage processors use the first step to receive messages sent in the previous stage, the second step to perform local computation, and the third step to send messages. A processor can be a *coordinator* or a *worker*. A phase may have multiple coordinators. The number of processors that assume the coordinator role is determined by the *martingale principle*: if none of the expected coordinators survive through the entire phase, then the number of coordinators for the next phase is doubled. If at least one coordinator survives in a given phase, then in the next phase there is only one coordinator. A phase that is completed with at least one coordinator alive is called *attended*, otherwise it is called *unattended*.

Processors become coordinators and balance their loads according to each processor’s *local view*. A local view contains the set of ids of the processors assumed to be alive. The local view is partitioned into *layers*. The first layer contains one processor id, the second two ids, the  $i^{\text{th}}$  contains  $2^{i-1}$  ids.

Given a phase, in the first stage, the processors perform a task according to the load-balancing rule derived from their local views and report the completion of the task to the coordinators of that phase (determined by their local views). In the second stage, the coordinators gather the reports, they update the knowledge of the done tasks and they multicast this information to the processors that are assumed to be alive. In the last stage, the processors receive the information sent by the coordinators and update their knowledge of done tasks and their local views. Given the full details of the algorithm, it is not difficult to see that the combination of coordinators and local views allows the processors to obtain the information that would be available from the oracle  $\mathcal{O}_L$  in the algorithm in Figure 1.

It was shown in [7] that the work of algorithm AN is  $W = O((t + p \log p / \log \log p) \log f)$  and its message complexity is  $M = O(t + p \log p / \log \log p + fp)$ , for  $f < p \leq t$ . The message complexity  $M$  is defined as the *total* number of point-to-point send by the processors during an execution of an algorithm.

The new failure-sensitive analysis of algorithm AN yields the following [18].

**THEOREM 2.3.** *Algorithm AN solves  $\text{Do-All}(t, p, f)$  with work  $W(t, p, f) = O(\log f(t + p \log p / \Lambda_{t,p,f}))$  and message complexity  $M(t, p, f) = O(t + p \log p / \Lambda_{t,p,f} + pf)$ .*

## Shared memory: Algorithm W

For the shared memory model with synchronous crash-prone processors, algorithm W of Kanellakis and Shvartsman [29] is the most work-efficient algorithm for  $\text{Write-All}$ . We give a brief description of the algorithm; additional details can be

found in [29, 28]. Algorithm W is structured as a parallel *loop* through four phases: (W1) a failure detecting phase, (W2) a load rescheduling phase, (W3) a work phase, and (W4) a phase that estimates the progress of the computation and the remaining work, and that controls the parallel loop. These phases use full binary trees with  $O(t)$  leaves. The processors traverse the binary trees top-down or bottom-up according to the phase. Each such traversal takes  $O(\log t)$  time (the height of a tree). For a single processor, each iteration of the loop is called a *block-step*; since there are four phases with at most one tree traversal per phase, each block step takes  $O(\log t)$  time.

In algorithm W the trees stored in shared memory serve as the gathering places for global information about the number of active processors, remaining tasks and load-balancing. It is not difficult to see that these binary trees indeed provide the information to the processors that would be available from the oracle  $\mathcal{O}_L$ , in the algorithm in Figure 1. The binary tree used in phase W2 to implement load-balancing and phase W3 to assess the remaining work is called the *progress tree*. The tasks are associated with the leaves of the progress tree. When a processor performs a task, it writes the value 1 to the corresponding leaf of the tree (initially all leaves have the value 0).

Prior analysis of work for algorithm W had it perform  $O(t + p \log t \log p / \log \log p)$  work for  $f < p \leq t$  [39, 50]. Note that this bound is conservative, since it does not include  $f$ , the number of crashes.

The new failure-sensitive approach [18] gives an improved and complete analysis of work for algorithm W.

**THEOREM 2.4.** *Algorithm W solves the  $\text{Write-All}(t, p, f)$  problem using work  $O(t + p \log t \log p / \Lambda_{t,p,f})$ .*

Note that the two algorithms [7, 29] are designed for different models and use dissimilar data and control structures, however both algorithms make their load-balancing decisions by gathering global knowledge. By understanding what work is expended on load-balancing vs. the inherent work overhead due to the lower bounds, we are able to obtain the failure-sensitive upper bounds.

## 2.3 Bounds for iterative algorithms

$\text{Write-All}$  algorithms have been used in developing simulations of failure-free algorithms on failure-prone processors. This is done by iteratively using a  $\text{Write-All}/\text{Do-All}$  algorithm to simulate the steps of  $t$  failure-free “virtual” processors on  $p$  failure-prone “physical” processors (the usual case is that the number of physical processors does not exceed the number of virtual processors, i.e.,  $p \leq t$ ). We abstract this idea as the *iterative Do-All* problem as follows:

The  $r$ -iterative  $\text{Do-All}$  problem, denoted  $r\text{-Do-All}(t, p, f)$ , is the problem of using  $p$  processors to solve  $r$  instances of  $t$ -task  $\text{Do-All}$  with the added restriction that every task of the  $i$ th instance must be completed before any task of the  $i + 1$ st instance is begun.

The oracle version  $r\text{-Do-All}^{\mathcal{O}}(t, p, f)$  is defined analogously. An obvious solution for this problem is to run a  $\text{Do-All}$  algorithm  $r$  times. If the work complexity of  $\text{Do-All}$  in a given model is  $W(t, p, f)$ , then the work of  $r\text{-Do-All}$  is clearly no more than  $r \cdot W(t, p, f)$ . This does not take into account the fact that  $f$  crashes occur throughout the  $r$  iterations.

We present a substantially better analysis of work, denoted  $W(r, p, t, f)$ , for crash-prone processors. In particular we provide *matching* upper and lower bounds on work for  $r$ -Do-All $^{\mathcal{O}}(t, p, f)$ , where  $f < p \leq t$ , for specific ranges of failures.

By iteratively using the algorithm of Figure 1 and using our analysis for a single Do-All we obtain the following:

**THEOREM 2.5** ([18]). *The iterative  $r$ -Do-All $^{\mathcal{O}^L}(t, p, f)$  problem for  $f < p \leq t$  can be solved with work  $W(r, t, p, f) = O(r \cdot (t + p \log p / \Lambda_{t,p,f}^r))$ .*

By extending our analysis for a single Do-All we obtain:

**THEOREM 2.6** ([18]). *Given any algorithm that solves  $r$ -Do-All $^{\mathcal{O}}(t, p, f)$  for  $f < p \leq t$  and under any oracle  $\mathcal{O}$ , there exists an adversary that causes work  $W(r, t, p, f) = \Omega(r \cdot (t + p \log p / \Lambda_{t,p,f}^r))$ .*

The improvement in the above upper and lower bounds are twofold. First, the derivation of the bounds reflects the improved failure-sensitive analysis for a single Do-All. Additionally, we have  $\Lambda_{t,p,f}^r \geq \Lambda_{t,p,f}$  for any  $r$ , moreover  $\Lambda_{t,p,f}$  is a constant with respect to  $r$ . The result is that our bounds (Theorem 2.5) are asymptotically better than those obtained by computing the product of  $r$  and the (non-iterated) Do-All bounds (Theorem 2.1) (for  $p \leq t$ ).

By iteratively using algorithm AN  $r$  times and using the oracle-based analysis we show how to solve  $r$ -Do-All $(t, p, f)$  ( $p \leq t$ ) on synchronous message-passing crash-prone processors with the following work and message complexity:

**THEOREM 2.7** ([18]). *The  $r$ -Do-All $(t, p, f)$  problem for  $f < p \leq t$  can be solved with work*

$$W(r, t, p, f) = O\left(r \cdot \log\left(\frac{t}{r}\right) \cdot (t + p \log p / \Lambda_{t,p,f}^r)\right),$$

*and with message complexity*

$$M(r, t, p, f) = O\left(r \cdot (t + p \log p / \Lambda_{t,p,f}^r) + fp\right).$$

By iteratively using algorithm W [29]  $r$  times and using the failure-sensitive analysis we show how to solve  $r$ -Write-All $(t, p, f)$  on synchronous shared-memory crash-prone processors (for  $f < p \leq t$ ).

**THEOREM 2.8** ([18]). *The  $r$ -Write-All $(t, p, f)$  problem for  $f < p \leq t$  can be solved on synchronous crash-prone processors, using shared memory, with work*

$$W(r, t, p, f) = O\left(r \cdot (t + p \log t \log p / \Lambda_{t,p,f}^r)\right).$$

The above result yields tighter bounds than the previously known bounds of  $O(r \cdot (t + p \log t \log p / \log \log t))$  for deterministic PRAM ([13]) simulations using algorithm W together with the simulation techniques such as [32, 49]:

**THEOREM 2.9.** *Any parallel synchronous  $t$ -processor,  $r$ -time, shared-memory algorithm (PRAM) can be simulated on  $p$  crash-prone synchronous processors (for  $f < p \leq t$ ) with work  $O\left(r \cdot (t + p \log t \log p / \Lambda_{t,p,f}^r)\right)$ .*

### 3. COOPERATION IN THE ABSENCE OF COMMUNICATION

We now completely turn to the setting where the Do-All problem needs to be solved by distributed message-passing processors. 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. In this section 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.

To study aspects of the trade-off between communication and computation in distributed cooperative applications, we consider a variation on the usual Do-All problem:  $p$  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  $W = O(t \cdot p)$ , 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  $W = \Omega(t + p \log p)$ , cf. [28]. 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  $p - 2$  processors that are unable to communicate, and they collectively must perform exactly  $t \cdot (p - 2)$  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  $t/2$  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, they could schedule their work as follows: one processor performs the tasks in the order  $1, 2, \dots, t$ , the other in the order  $t, t - 1, \dots, 1$ . 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 per-

formed 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 [?]) 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.

The problem of assessing redundant work for distributed cooperation in the absence of communication was studied in [10]. Other approaches to dealing with limited communication include [47] and [15]. Dolev *et al.* [10] 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  $t$ . 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 [10] develops an effective strategy for managing the task execution redundancy and proves that the strategy provides each of the  $p \leq t$  processors with a schedule of  $\Theta(t^{1/3})$  tasks such that at most one task is performed redundantly by any two processors. In the rest of this section we summarize key recent results in this area.

### 3.1 Schedules, waste, and designs

In our abstract setting there are  $p$  processors that need to perform  $t$  independent and idempotent tasks. The processors have unique identifiers from the set  $[p] = \{1, \dots, p\}$ , and the tasks have unique identifiers from the set  $[t] = \{1, \dots, t\}$ . Initially each processor knows the tasks that need to be performed and their identifiers (otherwise no fault-tolerant distributed solution is possible).

A  $(p, t)$ -schedule is a tuple  $(\sigma_1, \dots, \sigma_p)$  of  $p$  permutations of the set  $[t]$ . When  $p = 1$  it is elided and we simply write  $t$ -schedule. A  $(p, t)$ -schedule immediately gives rise to a strategy for  $p$  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  $\sigma_i$ . Suppose now that some  $k$  of these processors, say  $q_1, \dots, q_k$ , should rendezvous at a time when the  $i$ th 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 3.1. If  $S$  is a  $(p, t)$ -schedule and  $(a_1, \dots, a_k) \in \mathbb{N}^k$ , the waste function for  $S$  is

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

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

Here (and throughout), if  $\phi : X \rightarrow Y$  is a function and  $S \subset X$ , we let  $\phi(S) = \{\phi(x) \mid x \in S\}$ . For a specific vector  $a = (a_1, \dots, a_k)$ ,  $\mathcal{W}_S(a)$  captures the worst-case number of redundant tasks performed by any collection of  $k$  processors when the  $i$ th 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  $S$  is a  $(p, t)$ -schedule then

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

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 3.3). 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., [25] 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 3.2. A  $\ell$ - $(v, k, \lambda)$  design is a family of subsets  $\mathcal{S} = (S_1, \dots, S_n)$  of the set  $[v]$  with the property that each  $|S_i| = k$  and any set of  $\ell$  elements of  $[v]$  is a subset of precisely  $\lambda$  of the  $S_i$ . (N.B. The subsets  $S_i$  are typically referred to as blocks.)

Observe that if  $\mathcal{S}$  is a  $\ell$ - $(v, k, \lambda)$  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  $\ell$  which contain  $T$ ; let  $U_i, i \in [v - (\ell - 1)]$ , denote these sets. By assumption, each  $U_i$  appears in exactly  $\lambda$  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  $2$ - $(p, k, \lambda)$  design consisting of  $t$  sets  $S_1, \dots, S_t$ . For each  $i \in [p]$ , let  $T_i = \{j \mid i \in S_j\}$ . Note now that for any  $i \neq j$ ,

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

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  $\Sigma = (\sigma_1, \dots, \sigma_n)$  be any sequence of permutations of  $[t]$  for which  $\sigma_i([a]) = T_i$ . It is clear that these form an  $(p, t)$ -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  $\sigma_i([p] \setminus [a])$ , 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  $\text{GF}(q)$  denote the finite field with  $q$  elements, where  $q$  is a prime power. Treating  $\text{GF}(q)^3$  as a vector space over  $\text{GF}(q)$ , the design will be given by the lattice of linear subspaces of  $\text{GF}(q)^3$ . It is easy to check that there are  $n = q^2 + q + 1$  distinct one dimensional subspaces of  $\text{GF}(q)^3$ , which we denote  $\ell_1, \dots, \ell_n$ . We say that two subspaces  $\ell_i$  and  $\ell_j$  are *orthogonal* if  $\forall u \in \ell_i, \forall v \in \ell_j, \langle u, v \rangle = \sum u_j v_j \pmod q = 0$ ; 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  $\ell_i$ , and that this forms a  $2-(q^2 + q + 1, q + 1, 1)$  design. See [25] 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  $\langle t_u^1, \dots, t_u^q \rangle$  where  $S_u = \{\ell_{t_u^i} \mid 1 \leq i \leq q + 1\}$ ; i.e., the one dimensional subspaces  $\ell_{t_u^i}, i = 1, \dots, q + 1$ , are precisely those orthogonal to  $\ell_u$ . 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 3.1.** *Let  $n = q^2 + q + 1$ , where  $q$  is a prime power. Then the sequences  $\mathcal{L}_n = \langle L_1, \dots, L_n \rangle$  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.*

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

### 3.2 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  $\Omega(t \cdot g)$ , even if each group of processors performs no more than the optimal number of  $\Theta(t)$  tasks. In the extreme case where all processors are isolated from the beginning, the work of any algorithm is  $\Omega(t \cdot p)$ , 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  $\Omega(t \cdot p)$  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  $t/2$  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  $p$  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  $t' \leq \lfloor t/p \rfloor$ . In particular, if  $p = t$ , 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  $t$ .

Now we proceed to the lower bound, which generalizes the second Johnson Bound [26] for the case when two processors execute *different* number of tasks prior to their rendezvous.

**THEOREM 3.2** ([45]). *Let  $\mathcal{P} = \langle \pi_1, \dots, \pi_p \rangle$  be a  $(p, t)$ -schedule and let  $0 \leq a \leq b \leq t$ . Then*

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

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

**COROLLARY 3.3** ([36]). *For  $t = p$ , if  $a > \sqrt{t-3/4} + \frac{1}{2}$  then any  $p$ -processor schedule of length  $a$  for  $t$  tasks has worst case pairwise waste at least 2.*

### 3.3 Random schedules

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

#### 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  $\pi_i$  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 is bounded for *all* long enough prefixes.

**THEOREM 3.4** ([36]). *Let  $\mathcal{R}$  be a system of  $p$  random schedules for  $t$  tasks constructed as above. Then with probability at least  $1 - \frac{1}{pt}$ ,  $\forall a, b$  such that  $7\sqrt{t} \ln(2pt) \leq a, b \leq t$ ,  $\mathcal{W}_{\mathcal{R}}(a, b) \leq \frac{ab}{t} + \Delta(a, b)$ , where  $\Delta(a, b) = 11\sqrt{\frac{ab}{t} \ln(2pt)}$ .*

Observe that Theorem 3.2 shows that  $(p, t)$ -schedules must have waste  $\mathcal{W}(a, a) = \Omega(a^2/t)$  (as  $p \rightarrow \infty$ ); 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 3.5** ([37]). *Consider the random schedule  $\mathcal{R}$  as given above. Then with probability at least  $1 - 1/p$ ,*

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

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

Note that again this bounds the distance of the  $k$ -waste from its expected value, which can be computed by inclusion-exclusion to be  $\sum_{s=2}^k (-1)^s \binom{k}{s} \frac{a^s}{t^{s-1}}$ . The proof, which we omit, proceeds by considering the martingale which exposes the  $i$ th 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.)

## 3.4 Derandomization via finite geometries

We now consider a method for derandomizing these schedules using the design discussed in Section 3.1.

### Schedules for $p = t$

We construct a system of schedules of length  $p$  by arranging tasks from the sequences of  $\mathcal{L}_p$  in a recursive fashion. (Recall that while the sequences of  $\mathcal{L}_p$  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  $\mathcal{L}_p$  indexed by a specific subspace  $L_u$ .

**LEMMA 3.6** ([37]). *Let  $\mathcal{L}_p = \langle L_1, \dots, L_p \rangle$  be the collection of sequences constructed in Theorem 3.1, and let  $L_u = \langle t_u^1, \dots, t_u^{q+1} \rangle$ ,  $1 \leq u \leq p$ . Then for any  $i \neq j$ , we have  $L_{t_u^i} \cap L_{t_u^j} = \{u\}$ .*

As a result of this lemma, there is *only a single repeated element* in the sequences  $L_{t_u^1}, L_{t_u^2}, \dots, L_{t_u^{q+1}}$ ; this element is  $u$ . This fact suggests the following construction of a system of schedules  $\mathcal{P}_p$ . Let  $P_u$ ,  $1 \leq u \leq p$ , be the sequence whose first element is  $u$ , and whose remaining elements are given by concatenating the  $q+1$  sequences  $L_{t_u^1}, \dots, L_{t_u^{q+1}}$  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  $(q+1)q+1 = p$ , each element of  $[p]$  must appear exactly once in each  $P_u$ ; these  $P_u$  thus give rise to a family of permutations  $\pi_u$ , where  $\pi_u(k)$  is the  $k$ th element of  $P_u$ . Let  $\mathcal{P}_p = (\pi_1, \dots, \pi_p)$ .

We conceptually divide the sequences  $P_u$  (associated with the permutations  $\pi_u$ ) 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 3.7** ([37]). *Let  $q$  be a prime power,  $p = q^2 + q + 1$ . Let  $a = 1 + iq$ ,  $b = 1 + jq$ ,  $0 \leq i, j \leq q+1$ . Then*

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

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

### Controlling waste for short prefixes

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

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  $\text{GF}(q)^3$ . 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  $\text{GF}(q)^3$ , it is not hard to show that  $G_p$  is  $(q+1)$ -regular. By Hall's theorem (see, e.g., [23]), there is always a perfect matching in a  $d$ -regular bipartite graph and note that such a matching yields a permutation  $\rho$  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 [24] that runs in time  $O(\sqrt{|U| + |V|} \cdot |E|) = O(p^2)$ .

We use  $\rho$  to construct the system of schedules  $\mathcal{G}_p$  such that the first segments are distinct. Specifically, given  $\mathcal{L}_p$ , the system of schedules  $\mathcal{G}_p = \langle \gamma_1, \dots, \gamma_p \rangle$  is defined as follows. For any  $1 \leq u \leq p$ , 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  $\gamma_u$  is the permutation associated with  $G_u$ .

**THEOREM 3.8** ([37]). *Let  $q$  be a prime power,  $p = q^2 + q + 1$ . Let  $a = 1 + iq$ ,  $b = 1 + jq$ ,  $0 \leq i, j \leq q + 1$ . Then:*

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

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.

## 4. WORK-COMPETITIVE SCHEDULING

Given that no algorithm is able to maintain low total work in the presence of communication failures that partition the system, we pursue competitive analysis of the Do-All problem. In particular we consider the *partitionable network* consisting of  $p$  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, [5] and the surveys in [44]. 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, the sets of tasks completed by two groups of processors when these groups are merged should 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 groups of processors merge who have collectively completed more than  $t$  tasks will necessitate wasted work. A processor may cease executing tasks *only* when it knows the results of all tasks. We refer to this version of the Do-All problem as Do-All\*.

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  $\Omega(t \cdot p)$  work. On the other hand, any reasonable algorithm should attain  $O(t)$  work in the case where all processors remain connected during the computation. Considering that *every* algorithm

performs poorly in the totally disconnected case, it seems reasonable to treat the problem as an on-line problem and pursue *competitive analysis* [46].

We consider the behavior of an algorithm in the face of an adversary (which is *oblivious* in the sense of [4]) 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  $G_1$ . 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  $p$  the number of processors, then such a computation pattern is a labeled and weighted directed acyclic graph, that we call a  $(p, t)$ -DAG:

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

- (i)  $\forall v \in V$ ,  $h(v) \leq p$ , and for any maximal path  $\mathbf{p} = (v_1, \dots, v_k)$  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”:

$$[t] = \dot{\bigcup}_{v: \text{in}(v)=0} g(v).$$

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

$$g(v) = \dot{\bigcup}_{(u,v) \in E} \phi((u, v)),$$

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

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

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

**EXAMPLE.** A sample  $(12, t)$ -DAG is shown in Figure 2. 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

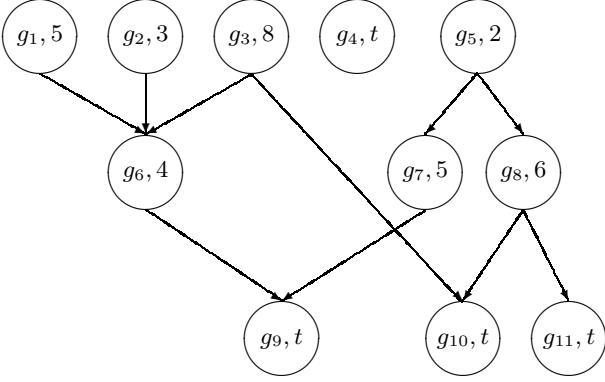


Figure 2: An example of a  $(12, t)$ -DAG

$g_{10}$  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.  $\square$

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 Do-All\* 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  $C$ .

We treat randomized algorithms as distributions over deterministic algorithms; for a set  $\Xi$  and a family of deterministic algorithms  $\{D_r \mid r \in \Xi\}$  we let  $R = \mathcal{R}(\{D_r \mid r \in \Xi\})$  denote the randomized algorithm where  $r$  is selected uniformly at random from  $\Xi$  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 4.2 ([46, 12, 4]). Let  $\alpha$  be a real valued function defined on the set of all  $(p, t)$ -DAGs (for all  $p$  and  $t$ ). A randomized algorithm  $R$  is  $\alpha$ -**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  $r \in \Xi$ .

We begin with a lower bound for *deterministic* algorithms. This is then applied to give a lower bound for randomized algorithms in Corollary 4.2.

THEOREM 4.1 ([19]). Let  $a : \mathbb{N} \rightarrow \mathbb{R}$  and  $D$  be a deterministic scheduling algorithm for Do-All\* so that  $D$  is  $a(\mathbf{cw}(\cdot))$ -competitive (that is  $D$  is  $\alpha$ -competitive, for a function  $\alpha = a \circ \mathbf{cw}$ ). Then  $a(c) \geq 1 + c/e$ .

This theorem is proved by producing a stochastic computation pattern  $C$  that is independent of the deterministic algorithm  $D$ —this immediately gives rise to a lower bound for randomized algorithms:

COROLLARY 4.2 ([19]). Let  $\mathcal{R}(\{D_r \mid r \in \Xi\})$  be a randomized scheduling algorithm for the Do-All\* 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  $\Pi = (\pi_1, \dots, \pi_p)$  be a  $p$ -tuple of permutations, where each  $\pi_i$  is a permutation of  $[t]$ . We describe a deterministic algorithm  $D_\Pi$  so that

$$\text{RS} = \mathcal{R}(\{D_\Pi \mid \Pi \in (S_{[t]})^p\}),$$

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  $D_\Pi$  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  $\pi_\gamma(1), \dots, \pi_\gamma(t)$  which is not in  $K$ .

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

THEOREM 4.3 ([19]). Algorithm RS is  $(1 + \mathbf{cw}(C)/e)$ -competitive for any computation pattern  $C$ .

From the definition of computation width, it is not difficult to observe that any computation pattern  $C$  containing only patterns of merges has  $\mathbf{cw}(C) = 1$ . Hence, from Corollary 4.2 and Theorem 4.3 we get that algorithm RS is optimally work-competitive for any given pattern of merges.

COROLLARY 4.4 ([19]). Algorithm RS is  $(1 + \frac{1}{e})$ -competitive for any  $(p, t)$ -DAG  $C$  with  $\mathbf{cw}(C) = 1$ .

## 5. DISCUSSION

We surveyed several recent results that characterize the ability of  $p$  processors to cooperate in performing a common set of  $t$  tasks in the presence of adversity. We presented failure-sensitive upper and lower bounds for distributed cooperation problems when synchronous processors are subject to crash failures. We presented bounds on redundant work for distributed cooperation during the prolonged absence of communication. Finally, we presented a competitive analysis of cooperative work performed by fragmenting and merging groups of processors.

There are several interesting open problems in this area. We mention just a few of them. For the asynchronous shared-memory model, a large gap remains between the lower bounds on work and the best known upper bound [2] for the full range of processors ( $p = t$ ). For both the shared-memory and message-passing models, there is some progress in reducing the upper bounds with the help of permutations that have certain combinatorial properties. Such permutations are known to exist, but it appears to be very challenging to construct them efficiently.

We established bounds on the competitive ratio of a natural randomized algorithm for scheduling in partitionable networks. One outstanding open question is to derandomize the schedules used by task-performing algorithms. Another promising direction is to study the task-performing paradigm in the models of computation that combine network reconfigurations with processor failures. The goal is to establish complexity results that show how performance of task-performing algorithms depends both on the extent of the network reconfiguration and on the number of processor failures.

**Credits.** The results surveyed in Sections 2 and 4 are contained in Georgiou's upcoming doctoral dissertation [16]. The results surveyed in Section 3 are contained in Malewicz's upcoming doctoral dissertation [34].

## 6. REFERENCES

- [1] N. Alon and J.H. Spencer. *The probabilistic method*. John Wiley & Sons Inc., 1992. With an appendix by Paul Erdős, A Wiley-Interscience Publication.
- [2] R.J. Anderson and H. Woll. Algorithms for the certified Write-All problem. *SIAM Journal of Computing*, 26(5):1277–1283, 1997.
- [3] Y. Aumann and M.O. Rabin. Clock construction in fully asynchronous parallel systems and PRAM simulation. In *Proc. of the 33<sup>rd</sup> IEEE Symposium on Foundations of Computer Science (FOCS 1992)*, pages 147–156, 1992.
- [4] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11(1):2–14, 1994.
- [5] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, 1993.
- [6] J. Buss, P.C. Kanellakis, P. Ragde, and A.A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 20(1):45–86, 1996.
- [7] B. Chlebus, R. De Prisco, and A.A. Shvartsman. Performing tasks on restartable message-passing processors. *Distributed Computing*, 14(1):49–64, 2001.
- [8] B.S. Chlebus, L. Gąsieniec, D.R. Kowalski, and A.A. Shvartsman. Bounding work and communication in robust cooperative computation. In *Proc. of the 16<sup>th</sup> International Symposium on Distributed Computing (DISC 2002)*, pages 295–310, 2002.
- [9] R. De Prisco, A. Mayer, and M. Yung. Time-optimal message-efficient work performance in the presence of faults. In *Proc. of the 13<sup>th</sup> ACM Symposium on Principles of Distributed Computing (PODC 1994)*, pages 161–172, 1994.
- [10] S. Dolev, R. Segala, and A.A. Shvartsman. Dynamic load balancing with group communication. In *Proc. of the 6<sup>th</sup> International Colloquium on Structural Information and Communication Complexity (SIROCCO 1999)*, pages 111–125, 1999 (to appear in *Theoretical Computer Science*).
- [11] C. Dwork, J. Halpern, and O. Waarts. Performing work efficiently in the presence of faults. *SIAM Journal on Computing*, 27(5):1457–1491, 1998. A preliminary version appears in *Proc. of the 11<sup>th</sup> ACM Symposium on Principles of Distributed Computing (PODC 1992)*, pages 91–102, 1992.
- [12] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.
- [13] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. of the 10<sup>th</sup> ACM Symposium on Theory of Computing (STOC 1978)*, pages 114–118, 1978.
- [14] Z. Galil, A. Mayer, and M. Yung. Resolving message complexity of byzantine agreement and beyond. In *Proc. of the 36<sup>th</sup> IEEE Symposium on Foundations of Computer Science (FOCS 1995)*, pages 724–733, 1995.
- [15] S. Georgiades, M. Mavronicolas, and P. Spirakis. Optimal, distributed decision-making: The case of no communication. In *Proc. of the 12<sup>th</sup> International Symposium on Fundamentals of Computation Theory (FCT 1999)*, pages 293–303, 1999.
- [16] Ch. Georgiou. *Robust Cooperative Computing*. Doctoral Dissertation, in preparation, University of Connecticut, 2003.
- [17] Ch. Georgiou, A. Russell, and A.A. Shvartsman. The complexity of distributed cooperation in the presence of failures. In *Proc. of the 4<sup>th</sup> International Conference on Principles of Distributed Systems (OPODIS 2000)*, pages 245–264, 2000.
- [18] Ch. Georgiou, A. Russell, and A.A. Shvartsman. The complexity of synchronous iterative Do-All with crashes. In *Proc. of the 15<sup>th</sup> International Symposium on Distributed Computing (DISC 2001)*, pages 151–165, 2001.
- [19] Ch. Georgiou, A. Russell, and A.A. Shvartsman. Work-competitive scheduling for cooperative computing with dynamic groups. In *Proc. of the 35<sup>th</sup> ACM Symposium on Theory of Computing (STOC 2003)*, to appear, 2003. (Preliminary results reported in the brief paper: Optimally work-competitive scheduling for cooperative computing with merging groups. In *Proc. of the 21<sup>st</sup> ACM Symp. on Principles of Distributed Computing (PODC 2002)*, 2002.)
- [20] Ch. Georgiou and A.A. Shvartsman. Cooperative computing with fragmentable and mergeable groups.

- Journal of Discrete Algorithms*, in press. A preliminary version appears in *Proc. of the 7<sup>th</sup> International Colloquium on Structural Information and Communication Complexity (SIROCCO 2000)*, pages 141–156, 2000.
- [21] J.F. Groote, W.H. Hesselink, S. Mauw, and R. Vermeulen. An algorithm for the asynchronous Write-All problem based on process collision. *Distributed Computing*, 14(2):75–81, 2001.
- [22] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5, pages 97–145. ACM Press/Addison-Wesley, 1993.
- [23] F. Harary. *Graph Theory*. Addison-Wesley, 1994.
- [24] J.E. Hopcroft and R.M. Karp. A  $O(n^{5/2})$  algorithm for maximum matching in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.
- [25] D.R. Hughes and F.C. Piper. *Design Theory*. Cambridge University Press, 1985.
- [26] S.M. Johnson. A new upper bound for error-correcting codes. *IEEE Transactions on Information Theory*, 8:203–207, 1962.
- [27] P.C. Kanellakis, D. Michailidis, and A.A. Shvartsman. Controlling memory access concurrency in efficient fault-tolerant parallel algorithms. *Nordic Journal of Computing*, 2(2):146–180, 1995.
- [28] P.C. Kanellakis and A.A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, 1997.
- [29] P.C. Kanellakis and A.A. Shvartsman. Efficient parallel algorithms can be made robust. *Distributed Computing*, 5(4):201–217, 1992. A preliminary version appears in the *Proc. of the 8<sup>th</sup> ACM Symposium on Principles of Distributed Computing (PODC 1989)*, pages 211–222, 1989.
- [30] Z.M. Kedem, K.V. Palem, M.O. Rabin, and A. Raghunathan. Efficient program transformations for resilient parallel computation via randomization. In *Proc. of the 24<sup>th</sup> ACM Symposium on Theory of Computing (STOC 1992)*, pages 306–318, 1992.
- [31] Z.M. Kedem, K.V. Palem, A. Raghunathan, and P. Spirakis. Combining tentative and definite executions for dependable parallel computing. In *Proc. of the 23<sup>rd</sup> ACM Symposium on Theory of Computing (STOC 1991)*, pages 381–390, 1991.
- [32] Z.M. Kedem, K.V. Palem, and P. Spirakis. Efficient robust parallel computations. In *Proc. of the 22<sup>nd</sup> ACM Symposium on Theory of Computing (STOC 1990)*, pages 138–148, 1990.
- [33] D. Kowalski and A. Shvartsman. Performing work with asynchronous processors: Message-delay-sensitive bounds. In *Proc. of the 22<sup>nd</sup> ACM Symposium on Principles of Distributed Computing (PODC 2003)*, to appear, 2003.
- [34] G. Malewicz. *Distributed Scheduling for Disconnected Cooperation*. Doctoral Dissertation, in preparation, University of Connecticut, 2003.
- [35] G. Malewicz. A work-optimal deterministic algorithm for the asynchronous Certified Write-All problem. In *Proc. of the 22<sup>nd</sup> ACM Symposium on Principles of Distributed Computing (PODC 2003)*, to appear, 2003.
- [36] G. Malewicz, A. Russell, and A.A. Shvartsman. Distributed cooperation during the absence of communication. In *Proc. of the 14<sup>th</sup> International Symposium on Distributed Computing (DISC 2000)*, pages 119–133, 2000.
- [37] G. Malewicz, A. Russell, A.A. Shvartsman. Optimal scheduling for disconnected cooperation. In *Proc. of the 8<sup>th</sup> International Colloquium on Structural Information and Communication Complexity (SIROCCO 2001)*, pages 259–274, 2001.
- [38] G. Malewicz, A. Russell, and A.A. Shvartsman. Local scheduling for distributed cooperation. In *Proc. of the 1<sup>st</sup> IEEE International Symposium on Network Computing and Applications (NCA 2001)*, 2001.
- [39] C. Martel. Personal communication. March, 1991.
- [40] C. Martel, A. Park, and R. Subramonian. Work-optimal asynchronous algorithms for shared memory parallel computers. *SIAM Journal on Computing*, 21(6):1070–1099, 1992.
- [41] C. Martel and R. Subramonian. On the complexity of certified Write-All algorithms. *Journal of Algorithms*, 16(3):361–387, 1994.
- [42] C. Martel, R. Subramonian, and A. Park. Asynchronous PRAMs are (almost) as good as synchronous PRAMs. In *Proc. of the 31<sup>st</sup> IEEE Symposium on Foundations of Computer Science (FOCS 1990)*, pages 590–599, 1990.
- [43] M. Momenzadeh, *Emulating shared-memory Do-All in asynchronous message passing systems*. Masters Thesis, Computer Sci. & Eng., University of Connecticut, 2003.
- [44] D. Powell, editor. *Special Issue on Group Communication Services*, volume 39(4) of *Communications of the ACM*. ACM Press, 1996.
- [45] A. Russell and A. A. Shvartsman. Distributed Computation Meets Design Theory: Local Scheduling for Disconnected Cooperation. *Bulletin of the EATCS*. 77:120–131, 2002.
- [46] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.
- [47] C.H. Papadimitriou and M. Yannakakis. On the value of information in distributed decision-making. In *Proc. of the 10<sup>th</sup> ACM Symposium on Principles of Distributed Computing (PODC 1991)*, pages 61–64, 1991.
- [48] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computing Systems*, 1(3):222–238, 1983.
- [49] A.A. Shvartsman. Achieving optimal CRCW PRAM fault-tolerance. *Information Processing Letters*, 39(2):59–66, 1991.
- [50] A.A. Shvartsman, *Fault-Tolerant and Efficient Parallel Computation*, Doctoral Dissertation, Computer Science, Brown University, Rhode Island, May 1992.