Performing Work in Distributed Systems

Notes: Initial notes gathered and cobbled by Alex A. Shvartsman

Today's reading:

- G. G. Malewicz, A. Russell, and A. A. Shvartsman. Distributed cooperation in the absence of communication. In Proceedings of the 14th International Conference on Distributed Computing, DISC, 2000. .ps
- C. Georgiou, A. Russell, and A.A. Shvartsman. The complexity of synchronous iterative do-all with crashes. Distributed algorithms (DISC 2001). .ps Next: TBD:

1 Introduction

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 N independent tasks* — is one of the standard problems in the research on the complexity of fault-tolerant distributed computation [DHW-SIAM98,KS-DC92]. Variations of this problem have been studied in shared-memory models (*Write-All*[1]) [KS97,KPRS-STOC91,MS-JA94], in message-passing models [DMY-PODC94,DHW-SIAM98,GMY-FOCS95], and in partitionable networks (*Omni-Do*) [DSS-SIR99,GS-JDA,MRS-DISC00]. 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.

In the design of practical distributed/parallel programs one needs to ensure good performance and dependability under unpredictable load patterns caused by deviations from synchrony or by the failures of some processors to complete tasks on time. Here again, a common challenge is to perform N independent tasks on P processors [HG-DC01]. Such tasks could be copying a large array, searching a collection of data, or applying a function to all elements of a matrix [GHVM-DC01].

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

[1]The *Do-All* problem and the *Write-All* problem are subtly different. The main difference is that in *Do-All* the tasks may be supplied to the processors from some external sources, while in *Write-All* the tasks are stored in shared-mmeory accessible to all processors. In some contexts it is possible to abstract away from the sources and the nature of the tasks and to treat *Do-All* and *Write-All* as the same problem. The distinction between the problems is reinstated when one applies the results of such abstract analysis to specific message-passing and shared-memory algorithms.

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 [DHW] 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 *at al.* [CGKS] 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.

2 The Write-All / Do-All Primitive

We define Do-All to be the following problem:

Definition 2.1 Given a set of N independent and idempotent tasks $A[1..N]$ and P processors, perform each task at least once

Solving Do-All when $P = N$ is one of the simplest computations that can be performed by a distributed system with P processors The following optimal synchronus program solves Do-All in constant time, thus achieving linear speed-up.

```
forall processors PID = 1..N parbegin
    perform A[PID]
parend
```

Next suppose that $P \neq N$. It easy to modify the program above so that each processor performs $\lceil N/P \rceil$ tasks. (Padding is used to include “empty” tasks when P does not divide N.)

```
forall processors PID= 1..P parbegin
    for i = 0 to ⌈N/P⌉ − 1 do
        perform A[PID+i * P]
    rof
parend
```

The above algorithm takes time $T_{N,P} = \Theta(N/P)$ and its cost (work) is $C = P \cdot T_{N,P} = P \cdot \Theta(N/P) = \Theta(N)$. Thus this solution is also optimal, and in particular, it achieves linear speed-up: $S = \Theta(N)/T_{N,P} = \Theta(N)/\Theta(N/P) = \Theta(P)$.

In shared-memory models, solving Do-All is equivalent (under some simple assumptions) to solving the following, more abstract problem, called Write-All.

Definition 2.2 Given a zero-valued array of N elements and P processors, the *Write-All* problem is to set each element of the array to 1.

The *Write-All* problem captures the computational progress that can be naturally accomplished in unit time by N synchronous processors in the absence of failures. Solving Do-All or Write-All efficiently in the presence of asynchrony and failures is very challenging.

Measuring efficiency. We are interested in studying the complexity of *Do-All* measured as *work* (cf. [KS-DC92,DHW-SIAM98,DMY-PODC94]). We assume that it takes a unit of time for a processor to perform a unit of work, and that a single task corresponds to a unit of work. Our definition of *work complexity* is based on the *available processor steps* measure [KS97]. Let $\mathcal{F}$ be the failure model. For a computation subject to a failure pattern F, $F \in \mathcal{F}$, where $\mathcal{F}$ is a failure model, denote by $P_i(F)$ the number of processors completing a unit of work in step i of the computation.

Definition 2.3 Given a problem of size N and a P-processor algorithm that solves the problem in the failure model $\mathcal{F}$, if the algorithm solves the problem for a pattern F in $\mathcal{F}$, with $|F| \leq f$, by time step $\tau(F)$, then the work complexity W of the algorithm is:

$$W_{N,P,f} = \max_{F \in \mathcal{F},\ |F| \leq f} \left\{ \sum_{i \leq \tau(F)} P_i(F) \right\}.$$

The idling processors consume a unit of work per step even though they do not contribute to the computation. Definition 2.3 does not depend on the specifics of the target model of computation, e.g., whether it is message-passing or shared-memory. (Communication complexity is defined similarly in Section **??**.)

Some research on Do-All in message-passing models uses a different definition of work, where processors are not charged for time spent idling or waiting for a message [DHW-SIAM98].

Do-All in message-passing models

Perform a set of N tasks using P fault-prone processors.

This problem was considered for the message-passing model by Dwork, Halpern and Waarts in their pioneering work [DHW92]. They developed several efficient algorithms for *Do-All* in the setting where the processors are subject to fail-stop (or crash) failures and where the tasks can be performed using the *at-least-once* execution semantics (i.e., the tasks can be made idempotent). Their model of computation assumes a synchronous system of processors, such that in one time unit a processor can perform a local computation and complete one task, and perform one round of communication consisting of sending and receiving messages.

Algorithmic techniques for distributed *Do-All* are motivated with the help of an observation that there is a qualitative duality between the shared-memory and the message-passing settings. In a shared-memory algorithm, a key datum, say reflecting an important component of the state of the computation, can be stored in a shared memory cell, and P processors can learn the value stored by performing P reads. In a message-passing algorithm, the only way to propagate information, stored anywhere, to P processors, is to do so by means of at least $P-1$ messages. Thus in message-passing algorithms, one sees a multicast or diffusion of a value to a subset of processors as a dual of these processors reading the value from a location in shared memory.

For the distributed message-passing models, one has to consider the costs of a computation in terms of the total work performed by the processors, and in terms of the number of messages sent by the processors. In the shared-memory models we have covered thus far, the algorithm efficiency was determined primarily by the *available processor steps* (or work) measure. Since in the worst case each instruction involves a constant number of memory accesses, work is also the measure of the number of memory accesses (i.e., the communication cost) performed by the computation. The one-to-one cost correspondence between work and communication does not apply to the message-passing models, since the number of messages sent and received by one processor in one round of communication can be 0, or it can be comparable to P when broadcasts are enabled.

To measure the communication efficiency of distributed algorithms, the number of messages sent by the processors is counted. One can also consider the total number of bits sent when the messages can vary in size or can be large. Measuring the work of a computation depends on the interpretation of the relative costs associated with different processing steps. Specifically, one needs to consider the costs associated with local processing and the costs associated with doing the tasks in solving the *Do-All* problem.

The efficiency of message-passing algorithms is characterized in terms of their work and message complexity. We define message complexity similarly to Definition 2.3 of work: For a computation subject to a failure pattern F, $F \in \mathcal{F}$, denote by $M_i(F)$ the number of point-to-point messages sent during step i of the computation. For a given problem of size N, if the computation solves the problem by step $\tau(F)$ in the presence of the failure pattern F, where $|F| \leq f$, then the message complexity M is:

$$M_{N,P,f} = \max_{F\in\mathcal{F},\ |F|\leq f} \left\{ \sum_{i\leq\tau(F)} M_i(F) \right\}.$$

3 Dwork Halpern Waarts

Optimizing Task-Oriented Work

In the setting of Dwork, Halpern and Waarts [DHW92], the local computation, whether performing low-level administrative tasks (such as maintaining the local clock) or idling, is considered to be negligible compared to the costs of performing the *Do-All* tasks, where processors may need to do constant size, but significant processing, or the costs of communication with external systems. This approach is consistent with the accepted accounting of costs in distributed algorithms. Additionally, Dwork, Halpern and Waarts consider *processes*, a notion that can be more general than *processors*. For example, when we interpret processes as *threads of control*, such that several threads may co-exist on the same processor, then the idle "steps" taken by threads need not be accounted for, since during such idle steps the processor hosting the thread can be executing other steps on behalf of other threads. Thus idle thread steps do not constitute a waste of the computational resources, and it is reasonable to account only for work spent on the *Do-All* tasks.

Dwork, Halpern and Waarts define the *effort* of an algorithm as the sum of its work and message complexity. This is done under the assumption that the cost of performing a unit of work and sending a message are comparable. They observe that at least $P+N-1$ units of work are necessary to solve the *Do-All* problem of size N using P processors. The goal then is the development of algorithms that perform $O(N + P)$ work, thus achieving work-optimality, while keeping the total effort reasonable. In this synchronous model, the running time of an algorithm is also a meaningful measure of efficiency, and in the analysis of algorithms this measure is also considered. The first two algorithmsre sequential — at any point in time only one task is being performed, thus in these algorithms the redundancy of having multiple processors contributes to fault-tolerance, but not to speed-up. The third algorithm achieves fault-free time optimality, with graceful degradation of running time in the presence of failures.

One algorithm presented by the authors (protocol $\mathcal{B}$) has effort $O(N + P\sqrt{P})$, with work contributing the cost $O(N + P)$ towards the effort, and message complexity contributing the cost $O(P\sqrt{P})$. The running time of the algorithm is $O(N + P)$. The use of synchrony in the algorithm is limited to failure detection by means of time-outs. Such algorithms can be converted to asynchronous algorithms when the underlying system is equipped with failure detectors. In this algorithm the N tasks are divided into *chunks* and each of these is divided into *subchuncks*. Processors checkpoint their progress by multicasting the completion information to subsets of processors after performing a subchunk, and broadcasting to all processors after completing chunks of work.

Another algorithm in [DHW92] (protocol $\mathcal{C}$) has effort $O(N + P \log P)$. This includes optimal work of $O(N + P)$, message complexity of $(P \log P)$, and time $O(P^2(N + P)2^{N+P})$. Thus the reduction in message complexity is traded-off for a significant increase in time.

The third work-optimal algorithm (protocol $\mathcal{D}$) is designed for maximum speed-up, which is achieved with a more aggressive checkpointing strategy, thus trading-off time for messages. The message complexity is quadratic in P for the fault-free case, and in the presence of a failure pattern F, the message complexity degrades to $\Theta(|F| \cdot P^2)$.

One interesting application of the *Do-All* algorithms is to solving Byzantine agreement with crash failures. Dwork, Halpern and Waarts show that a *Do-All* algorithm with P processors and N tasks can be used to solve agreement for N processors and less than P failures [DHW92]. In this application, the tasks performed by processors are sending messages to other processors informing them of the agreement value.

4 Chlebus De Prisco Shvartsman

Two new algorithms are presented and analyzed: AN and AR. All previous algorithms do not consider the possibility that a faulty processor is repaired and reintegrated into the system. [CDS] introduces a new algorithmic technique based on an aggressive coordination paradigm that permits multiple concurrent coordinators. This approach is suggested by the earlier observation that algorithms with only one coordinator cannot deal efficiently with restarts. The number of coordinators is managed adaptively. When failures of coordinators disrupt the progress of the computation, the number of coordinators is increased; when the failures subside, a single coordinator is appointed.

4.1 Algorithm AN

Algorithm AN presented by Chlebus *et al.* [CDS-DC01] uses a multiple-coordinator approach to solve *Do-All*(N, P, f) on crash-prone synchronous message-passing processors ($P \leq N$). The model assumes that messages incur a known bounded delay and that reliable multicast [HT93] is available, however messages to/from faulty processors may be lost.

		W: available processor steps (work)	M: message complexity
No restarts ($f < p$)	[DMY]	$O(t + (f+1)p)$	$O((f+1)p)$
	[GMY]	$O(t + (f+1)p)$	$O(fp^{\varepsilon} + \min\{f+1, \log p\}p)$
	AN	$O((t + p\log p/\log\log p)\log f)$	$O(t + p\log p/\log\log p + fp)$
Restarts ($f < p+r$)	AR	$O((t + p\log p + f)\cdot\min\{\log p, \log f\})$	$O(t + p\log p + fp)$

Figure 1: Efficiency of the solutions in [DMY,GMY] and algorithms AN and AR (the solutions in [DHW] consider a different notion of work complexity and focus on evaluation of effort).

4.1.1 Description of the Algorithm

We now give a brief description of the algorithm; additional details are given in Appendix A (but to avoid a complete restatement, we refer the reader to [CDS-DC01]). 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^{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.

It is shown in [CDS-DC01] that the work of algorithm AN is $W = O((N + N\log N/\log\log N)\log f)$ and its message complexity is $M = O(N + P\log P/\log\log P + fP)$, for $P \leq N$.

In the rest of this section we present the new analysis of work and message complexity of algorithm AN. Throughout we assume that the algorithm correctness is shown as in [CDS-DC01].

4.1.2 Algorithmic details

The following information about algorithm AN is taken from [CDS-DC01]. More information can be obtained there. The algorithm proceeds in a *loop* which is repeated 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 (thus a phase consists of 9 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. We refer to these three step as the *receive* substage, the *compute* substage and the *send* substage.

Coordinators and workers. A processor can be a *coordinator* of a given phase. All processors (including coordinators) are *workers* in a given phase. Coordinators are responsible for recording progress, while workers perform tasks and report on that to the coordinators. In the first phase one processor acts as the coordinator. There may be multiple coordinators in subsequent phases. 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. Whenever at least one coordinator survives a given phase, the number of coordinators for the next phase is reduced to one.

If at least one processor acts as a coordinator during a phase and it completes the phase without failing, we say that the phase is *attended*, the phase is *unattended* otherwise.

Local views. Processors assume the role of coordinator based on their local knowledge. During the computation each processor w maintains a list $L_w = \langle q_1, q_2, ..., q_k \rangle$ of supposed live processors. We call such list a *local view*. The processors in L_w are partitioned into *layers* consisting of consecutive sublists of L_w: $L_w = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle$. The number of processors in layer Λ^{i+1}, for $i = 0, 1, ..., j-1$, is the double of the number of processors in layer Λ^i. Layer Λ^j may contain less processors. When $\Lambda^0 = \langle q_1 \rangle$ the local view can be visualized as a binary tree rooted at processor q_1, where nodes are placed from left to right with respect to the linear order given by L_w. Thus, in a tree-like local view, layer Λ^0 consists of processor q_1, layer Λ^i consists of 2^i consecutive processors starting at processor q_{2^i} and ending at processor $q_{2^{i+1}-1}$, with the exception of the very last layer that may contain a smaller number of processors. Processors in a local view do not necessarily appear in the order of processor identifiers.

The local view is used to implement the martingale principle of appointing coordinators as follows. Let $L_{\ell,w} = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle$ be the local view of worker w at the beginning of phase ℓ. Processor w expects processors in layer Λ^0 to coordinate phase ℓ; if no processor in layer Λ^0 completes phase ℓ, then processor w expects processors in layer Λ^1 to coordinate phase $\ell+1$; in general processor w expects processors in layer Λ^i to coordinate phase $\ell+i$ if processors in all previous layers Λ^k, $\ell \le k < \ell + i$, did not complete phase $\ell + k$. The local view is updated at the end of each phase.

Phase structure and task allocation. The structure of a phase of the algorithm is as follows. Each processor w keeps its local information about the set of tasks already performed, denoted D_w, and the set of live processors, denoted P_w, as known by processor w. Set D_w is always an underestimate of the set of tasks actually done and P_w is always an overestimate of the set of processors that are "available" from the start of the phase. We denote by U_w the set of *unaccounted* tasks, i.e., whose done status is unknown to w. Sets U_w and D_w are related by $U_w = \mathcal{T} \setminus D_w$, where $\mathcal{T}$ is the set of all the tasks. Given a phase ℓ we use $P_{\ell,w}$, $U_{\ell,w}$ and $D_{\ell,w}$ to denote the values of the corresponding sets at the beginning of phase ℓ.

Computation starts with phase 0 and any processor q has all processors in $L_{0,q}$ and has $D_{0,q}$ empty. At the beginning of phase ℓ each worker (that is, each processor) w performs one task according to its local view $L_{\ell,w}$ and its knowledge of the set $U_{\ell,w}$ of unaccounted tasks, using the following *load-balancing rule*. Worker w executes the task whose rank is $(i \texttt{ mod } |U_{\ell,w}|)^{th}$ in the set $U_{\ell,w}$ of unaccounted tasks, where i is the rank of processor w in the local view $L_{\ell,w}$. Then the worker reports the execution of the task to all the processors that, according to the worker's local view, are supposed to be coordinators of phase ℓ. For simplicity we assume that a processor sends a message to itself when it is both worker and coordinator. Any processor c that, according to its local view, is supposed to be coordinator, gathers reports from the workers, updates its information about $P_{\ell,c}$ and $U_{\ell,c}$ and broadcasts this new information causing the local views to be reorganized. In [CDS-DC01] it is shown that at the beginning of any phase ℓ all live processors have the same local view L_ℓ and the same set U_ℓ of unaccounted tasks and that accounted tasks have been actually executed. A new phase starts if U_ℓ is not empty.

The following is a detailed description of a phase.

Phase ℓ of algorithm AN:

STAGE 1. The receive substage is not used. In the compute substage, any processor w performs a specific task z according to the load-balancing rule. In the send substage processor w sends a `report`(z) to any coordinator, that is, to any processor in the first layer of the local view $L_{\ell,w}$.

STAGE 2. In the receive substage the coordinators gather `report` messages. For any coordinator c, let $z_c^1, \ldots, z_c^{k_c}$ be the set of TIDs received. In the compute substage c sets $D_c \leftarrow D_c \cup \bigcup_{i=1}^{k_c}\{z_c^i\}$, and P_c to the set of processors from which c received `report` messages. In the send substage, coordinator c multicasts the message `summary`(D_c, P_c) to processors in P_c.

STAGE 3. During the receive substage `summary` messages are received by live processors. For any processor w, let $(D_w^1, P_w^1), \ldots, (D_w^{k_w}, P_w^{k_w})$ be the sets received in `summary` messages In the compute substage w sets $D_w \leftarrow D_w^i$ and $P_w \leftarrow P_w^i$ for an arbitrary $i \in \{1, \ldots, k_w\}$ and updates its local view L_w as described below. The send substage is not used.

Local view update rule. In phase 0 the local view $L_{0,w}$ of any processor w is a tree-like view containing all P processors ordered by their PIDs. Let $L_{\ell,w} = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle$ be the local view of processor w for phase ℓ. We distinguish two possible cases.

CASE 1. Phase ℓ is unattended. Then the local view of processor w for phase $\ell + 1$ is $L_{\ell+1,w} = \langle \Lambda^1, ..., \Lambda^j \rangle$.

CASE 2. Phase ℓ is attended. Then processor w receives `summary` messages from some coordinator in Λ^0. Processor w computes its set P_w as described in stage 3 (we will see that all processors compute the same set P_w). The local view $L_{\ell+1,w}$ of w for phase $\ell + 1$ is a tree-like local view containing the processors in P_w ordered by their PIDs.

Finally, we demonstrate how a computation of algorithm AN proceeds. Given an execution of the algorithm, we enumerate the phases. We denote the attended phases of the execution by $\alpha_1, \alpha_2, \ldots$, etc. We denote by π_i the sequence of unattended phases between the attended phases α_i and α_{i+1}. We refer to π_i as the i^{th} (unattended) period; an unattended period can be empty. Hence the computation proceeds as follows: unattended period π_0, attended phase α_1, unattended period π_1, attended phase α_2, and so on.

4.2 Algorithm AR

Algorithm AR is similar to algorithm AN; the difference is that there are added messages to handle the restart of processors. After the restart, processor q broadcasts `restart`(q) messages in each step until it receives a response. Processors receiving such messages, ignore them if these messages are not received in the receive substage of stage 2 of a phase. Thus we can imagine that a restarted processor q broadcasts a `restart`(q) in the send substage of stage 1 of a phase ℓ (however we will count all the `restart` messages in the message complexity). This message is then received by all the live and restarted processors of that phase, and, as we will see shortly, processor q is re-integrated in the view for phase $\ell + 1$. Processor q needs to be informed about the status of the ongoing computation. Hence processors that have this information send the `info`(U_ℓ, L_ℓ) messages to processor q with the set U_ℓ of unaccounted tasks and the local view L_ℓ. Next we provide the detailed description for each phase. The parts that are new or that are different in algorithm AR as compared to algorithm AN are *italicized*.

Phase ℓ of algorithm AR:

STAGE 1. The receive substage is not used. In the compute substage any processor w performs a specific task z according to the load balancing rule. In the send substage w sends a `report`(z) to any coordinator, that is, to any processor in the first layer of $L_{\ell,w}$. *Any restarted processor q broadcasts the* `restart`*(q) message informing all live processors of its restart.*

STAGE 2. In the receive substage the coordinators gather `report` messages *and all processors gather* `restart` *messages. Let R be the set of processors that sent a* `restart` *message.* For any coordinator c, let $z_c^1, ..., z_c^{k_c}$ be the set of TIDs received in `report` messages. In the compute substage c sets $D_c \leftarrow D_c \cup \bigcup_{i=1}^{k_c} \{z_c^i\}$ and P_c to the set of processors from which c received `report` messages. In the send substage, coordinator c multicasts the message `summary`(D_c, P_c) to the processors in P_c *and R. Any processor in P_c sends the message* `info`*(U_ℓ, L_ℓ) to processors in R.*

STAGE 3. *In the receive substage processors in R receive* `info`*(U_ℓ, L_ℓ) messages and processors in P_c and R receive* `summary`*(D_c, P_c) messages. In the compute substage, a restarted processor q sets $L_{\ell,q} \leftarrow L_\ell$ and $U_{\ell,q} \leftarrow U_\ell$.* Let $(D_w^1, P_w^1), ..., (D_w^{k_w}, P_w^{k_w})$ be the sets received in `summary` messages by processor w. Processor w sets $D_w \leftarrow D_w^i$ and $P_w \leftarrow P_w^i$ for an arbitrary $i \in 1, ..., k_w$ and updates its local view $L_{\ell,w}$ *as described below.* The send substage is not used.

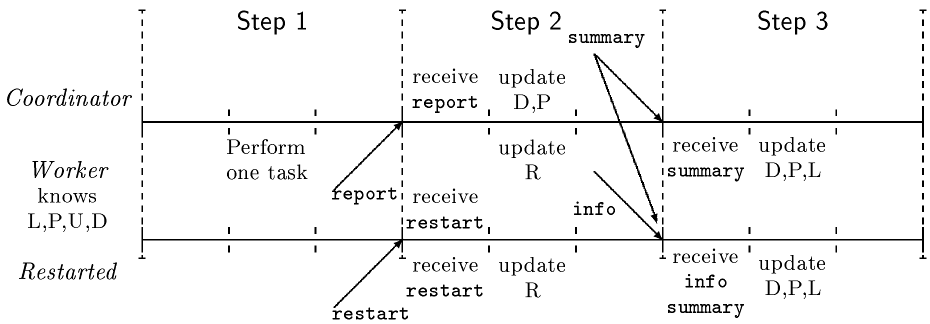

Figure 2: A phase of algorithm AR (for algorithm AN ignore the bottom line, which represents restarted processors, and all the messages referring to it).

Loal view update rule. In phase 0 the local view $L_{0,w}$ of any processor w contains all the processors in ¶ ordered by their PIDs, and the first layer is a singleton set. Let $L_{\ell,w} = \langle \Lambda^0, \Lambda^1, ..., \Lambda^j \rangle$ be the local view of processor w for phase ℓ. We distinguish two possible cases.

CASE 1. Phase ℓ is unattended. Let R^ℓ be the set of restarted processors which send `restart` messages. Let R' be the set of processors of R^ℓ that are not already in the local view $L_{\ell,w}$. Let $\langle R' \rangle$ be the processors in R' ordered according to their PIDs. The local view for the next phase is $L_{\ell+1,w} = \langle \Lambda^1, ..., \Lambda^j \rangle \oplus \langle R' \rangle$. The operator $\oplus$ places processors of R', in the order $\langle R' \rangle$, into the last layer Λ^j till this layer contains exactly the double of the processors of layer Λ^{j-1} and possibly adds a new layer Λ^{j+1} to accommodate the remaining processors of $\langle R' \rangle$. That is, newly restarted processors which are not yet in the view, are appended at the end of the old view. Notice that restarted processors, which receive `info` messages, know the old view L_ℓ.

CASE 2. Phase ℓ is attended. Let R^ℓ be the set of restarted processors. Since the phase is attended `summary` messages are received by all the live processors (including the restarted ones). Any processor w updates P_w as described in stage 3. Processor w knows the set R^ℓ. The local view $L_{\ell+1,w}$ for the next phase is structured according to the martingale principle and contains all the processors in $P_w \cup R^\ell$ ordered according to their PIDs.

If there are no restarts, algorithm AR behaves as algorithm AN. Figure 2 provides a graphical description of both algorithms.

5 Georgiou Russell Shvartsman

We let *Do-All*(N, P, f) stand for the *Do-All* problem for N tasks, P processors and up to f failures. We let *Do-All*$^{\mathcal{O}}(N, P, f)$ denote the *Do-All*(N, P, f) problem that is solved with the use of an arbitrary, omniscient oracle $\mathcal{O}$ that may make available to the processors any 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 message-passing or shared memory models as the oracle may make available to each processor the message or memory-read history it would have observed, *(ii)* any algorithm in the oracle model may be simulated in a message-passing or shared memory model by suitably simulating the oracle. As most *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 use exactly this approach to derive new f-sensitive upper bounds for message-passing and shared-memory models.

One may study the complexity of *Do-All*$^{\mathcal{O}}(\cdot)$ in conjunction with a variety of different oracles. When we prove lower bounds, we consider arbitrary oracles that may make available, for example, the entire global history of the computation on every processor. When we prove 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*$_i$, 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 $i_1, \ldots, i_k \in [P]$ are alive and tasks $j_1, \ldots, j_\ell \in [N]$ are incomplete at the beginning of the iteration, then *Oracle-task*$_{i_s} = j_t$, where $t = (s - 1 \bmod \ell) + 1$.

Prior to [GRS] no comprehensive failure-sensitive analysis of Do-All was known. Yet practical concerns would be well served by the knowledge of what happens as the number of failures grows from 1 to $P-1$. In particular, it is important to understand the behavior of the best algorithms for the entire range of f.

The detailed contributions in [GRS] are as follows.

I. We provide upper bounds (Section **??**) and matching lower bounds (Section 7) that are f-sensitive, specifically we give a *complete* analysis of $Do\text{-}All^{\mathcal{O}}(N,P,f)$ for the *entire* range of f. The bounds on work W are:[2]

$$\begin{array}{lll} (a) & W = \{O|\Omega\}\left(N + P + P\frac{\log(\min(N,P))}{\log(P/f)}\right) & \text{when } f \le \frac{P}{\log(\min(N,P))}, \\ (b) & W = \{O|\Omega\}\left(N + P + P\frac{\log(\min(N,P))}{\log\log(\min(N,P))}\right) & \text{when } f > \frac{P}{\log(\min(N,P))}. \end{array} \quad (1)$$

The lower bounds of course apply to algorithms in weaker models (e.g., a message passing or shared memory model).

In the rest of the paper we use a shorthand notation used to combine the statements such as (1)(a) and (1)(b) above. We define the quantity $\Lambda^r_{N,P,f}$, where r is the number of *Do-All* iterations (for *r-iterative Do-All*), and where N, P, and f appear in their usual roles, as follows.

$$\Lambda^r_{N,P,f} = \begin{cases} \log(\frac{Pr}{f}) & \text{when } f \le \frac{Pr}{\log(\min(N,P))}, \\ \log\log(\min(N,P)) & \text{when } f > \frac{Pr}{\log(\min(N,P))}. \end{cases} \quad (2)$$

For a single instance *Do-All*, i.e., when $r = 1$, we simplify this by defining $\Lambda_{N,P,f} = \Lambda^1_{N,P,f}$. Thus the result (1) is succinctly given by the following.

$$W = \{O|\Omega\}\left(N + P + P\log(\min(N,P))/\Lambda_{N,P,f}\right). \quad (3)$$

We use our bounds (3) (or equivalently (1)) to derive new bounds for algorithms where the extant analyses do 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. We show the following.

II. In Section **??** we provide new analysis of algorithm AN of Chlebus *et al.* [CDS-DC01] for *Do-All* in the message-passing model with crashes ($P \le N$). This algorithm has best known work for moderate number of failures. We show the complete analysis of work W and message complexity M:

$$W = O\left(\log f\left(N + P\log P/\Lambda_{N,P,f}\right)\right) \quad \text{and} \quad M = O\left(N + P\log P/\Lambda_{N,P,f} + Pf\right).$$

[2] We let "$\{O|\Omega\}$" stand for "O" when describing an upper bound and for "Ω" when describing a lower bound. All logarithms are to the base 2 unless explicitly specified otherwise. It is understood, that when $f=0$, then $X/\log(Y/f) = 0$, for any $X \ne 0$ and $Y \ne 0$.

III. In Section **??** we give a *complete* analysis of the work complexity W of the algorithm of Kanellakis and Shvartsman [KS-DC92] that solves the *Write-All* problem in synchronous shared-memory systems (shared memory *Do-All*) with processor crashes ($P \leq N$):

$$W = O\left(N + P \log N \log P / \Lambda_{N,P,f}\right).$$

Note that the two algorithms [CDS-DC01,KS-DC92] 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 (1), we are able to obtain the new results, and to demonstrate the utility and the generality of our approach. (We restrict our attention to the case $P \leq N$ for consistency with [CDS-DC01,KS-DC92].)

Do-All algorithms have been used in developing *simulations* of failure-free algorithms on failure-prone processors, e.g., [KPS-STOC90,AS91]. This is done by iteratively using a *Do-All* algorithm to simulate the steps of the N failure-free "virtual" processors on P failure-prone "physical" processors (here the usual case is that the number of physical processors does not exceed the number of virtual processors, i.e., $P \leq N$). In this paper we abstract this idea as the *iterative Do-All* problem as follows:

> The r-*iterative Do-All* problem, denoted r-*Do-All*(N, P, f), is the problem of using P processors to solve r instances of N-task *Do-All* with the added restriction that every task of the ith instance must be completed before any task of the $i+1$st instance is begun.

The oracle version r-*Do-All*$^{\mathcal{O}}(N, P, f)$ is defined analogously. An obvious solution for this problem is to run a *Do-All* algorithm r times. If the work complexity of *Do-All* in a given model is $W_{N,P,f}$, then the work of r-*Do-All* is clearly no more than $r \cdot W_{N,P,f}$. This does not take into account the fact that f crashes occur throughout the r iterations. We present a substantially better analysis for crash-prone processors:

IV. In Section **??** we show matching upper and lower bounds on work W for r-*Do-All*$^{\mathcal{O}}(N, P, f)$, where $f < P \leq N$, for specific ranges of failures.

$$W = \{O|\Omega\}\left(r \cdot \left(N + P \log P / \Lambda^r_{N,P,f}\right)\right). \tag{4}$$

The improvement in the bounds (4) are twofold. First, the derivation of the bounds reflects our improved failure-sensitive analysis for a single *Do-All*. Additionally, we have $\Lambda^r_{N,P,f} \geq \Lambda_{N,P,f}$ for any r, moreover $\Lambda_{N,P,f}$ is a constant with respect to r. The result is that our bounds (4) are asymptotically better than those obtained by computing the product of r and the (non-iterated) *Do-All* bounds (1) (for $P \leq N$).

V. In Section **??** we show how to solve r-*Do-All*(N, P, f) ($P \leq N$) on synchronous message-passing processors with the following work (W) and message (M) complexity.

$$W = O\left(r \cdot \log(\tfrac{f}{r}) \cdot \left(N + P \log P / \Lambda^r_{N,P,f}\right)\right) \text{ and } M = O\left(r \cdot \left(N + P \log P / \Lambda^r_{N,P,f}\right) + fP\right).$$

VI. In Section **??** we use *r-Do-All*(N,P,f) $(P \leq N)$ to show that P crash-prone processors can simulate any synchronous N-processor, r-time shared-memory algorithm (PRAM) with work:

$$W = O\left(r \cdot \left(N + P \log N \log P / \Lambda^r_{N,P,f}\right)\right).$$

This last result is strictly better than the previously known deterministic bounds for deterministic parallel algorithm simulations using the *Write-All* algorithm [KS-DC92] (the best known to date) together with the simulation techniques such as [KPS-STOC90,AS91].

6 Malewicz Russell Shvartsman (00, 01, 02)

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. [KS]. 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 [CACM96]) 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.

For a variation of the problem we deal with in this report, Dolev *et al.* [DSS] 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 [DSS] 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.

6.1 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 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 6.1 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

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

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 7.1.2. 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., [**?**] 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 6.2 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 $i \neq j$,

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

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 $\Sigma = (\sigma_1,\ldots,\sigma_n)$ 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 [**?**] 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 6.3 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.)

7 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 $\Omega(t \cdot n)$, 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 proceed to the lower bound, which generalizes the second Johnson Bound [**?**] for the case when two processors execute *different* number of tasks prior to their rendezvous.

Theorem 7.1 **([MRS-DISC])** Let $\mathcal{P} = \langle \pi_1, \dots, \pi_n \rangle$ be an (n,t)-schedule and let $0 \le a \le b \le t$. Then

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

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 7.2 **([MRS-DISC])** 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.

7.1 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 $[t]$.

7.1.1 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 7.3 **([MRS-DISC])** Let $\mathcal{R}$ be a system of n random schedules for t tasks constructed as above. Then with probability at least $1 - \frac{1}{nt}$,

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

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

Observe that Theorem 7.1 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.

7.1.2 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 7.4 Consider the random schedule $\mathcal{R}$ from above. Then with probability at least $1-1/n$,

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

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

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

7.1.3 Arbitrary rendezvous patterns (Georgiou Russell Shvartsman)

Thus far we have established bounds on wasted work for a single rendezvous. It is also interesting to understand the effectiveness of randomized schedules when encountering arbitrary patterns of (not necessarily pairwise) rendezvous. In general, an adversary that leaves the processors partitioned into g disconnected components 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. This lower bound forms a somewhat pessimistic landscape for our problem. Nevertheless, it is important to design and analyze algorithms that can be shown to be more efficient than the oblivious approach where each processor performs all tasks. In this case where the lower bound is quite sensitive to the interconnection pattern, we consider a *competitive* analysis [ST-CACM85] of randomized schedules.

Here we focus on the scenario where processors begin working (asynchronously) on the tasks in isolation and an adversary may choose to merge together various processors or groups of processors as the computation progresses. Once two processors find themselves in the same group, they cannot be separated–the adversary may never fragment a group of processors once it has been formed. An algorithm in this model is a rule which, given a collection of processors and a collection of tasks, determines the *next* task for this group to complete. A group of processors P who have collectively completed tasks T can use such a rule to determine the next task to work on. We shall consider the randomized algorithm, where a group of processors always selects a task at random from those it has not completed to work on next.

Adversaries in this model can be serialized in the following sense without loss of generality. An adversary A is described by a sequence of *events*; each event either *(i.)* allows a specific processor or group of processors to complete some number t of tasks, or *(ii.)* merges together some collection of processors and groups to form a new group. Note that when groups are formed by the adversary they inherit from their constituents full knowledge of all tasks they have previously completed.

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

Let D be a deterministic algorithm for our problem and A an adversary, we let $W_{(t,n)}(D,A)$ denote the total work (i.e., the number of tasks executed, counting multiplicities) of algorithm D with n

processors and t tasks, where the pattern of rendezvous is determined by adversary A. Where t and n can be inferred from context, we simply write $W(D, A)$.

We treat randomized algorithms as distributions over deterministic algorithms; for a set C and a family of deterministic algorithms $\{D_r \mid r \in C\}$ we let $R = \mathcal{R}a(\{D_r \mid r \in C\})$ denote the randomized algorithm where r is selected uniformly at random from C and scheduling is done according to D_r. We let OPT denote the optimal off-line algorithm, which may schedule tasks with full knowledge of the sequence of events caused by the adversary.

Definition 7.5 ([ST-CACM85,FKLMSY-JA91,BBKTW94])
A randomized algorithm R is α-**competitive** if for all adversaries A,

$$\mathbf{E}_r[W(D_r, A)] \leq \alpha \cdot W(OPT, A),$$

this expectation being taken over choice of $r \in \Omega$.

The following lower bound holds for the competitive ratios.

Theorem 7.6 ([MERGE-DO]) Let $\mathcal{R}a(\{D_r \mid r \in \Omega\})$ be an α-competitive randomized algorithm for our problem. Then $\alpha \geq 1 + \frac{1}{e}$.

This lower bound suggests that it is indeed worthwhile to seek algorithms that are very competitive, despite the potentially high bounds on "absolute" work. Armed with this hope we analyze the competitive performance of randomized schedules.

As mentioned, 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 $[t] \backslash K$. More formally, let $\Pi = (\pi_1, \ldots, \pi_n)$ be a (n, t)-schedule (where each π_i is a permutation of $[t]$). We describe a deterministic algorithm D_Π so that $RS = \mathcal{R}a(\{D_\Pi \mid \Pi \in (S_t)^n\})$. 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 π_γ which is not in K.

It turns out, not unexpectedly, that this algorithm is optimal with respect to the lower bound on competitive ratios.

Theorem 7.7 ([MERGE-DO]) Algorithm RS is $\left(1 + \frac{1}{e}\right)$-competitive for any given pattern of rendezvous.

Currently we have obtained preliminary results extending this competitive analysis to arbitrary patterns of merges and fragmentations.

7.2 Derandomization via finite geometries

We now consider a method for derandomizing these schedules using the design discussed in Section 6.1.

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 7.8 Let $\mathcal{L}_n = \langle L_1, \ldots, L_n \rangle$ be the collection of sequences constructed in Theorem 6.3, 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 $L_{t_u^i} \cap L_{t_u^j} = \{u\}$.

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 7.9 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

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

We mention that the construction can be done on-line. For each schedule the first element can be calculated in $O(1)$ time. For the remaining $q(q+1)$ elements, at the beginning of every sequence of q elements we need to invert at most two elements in 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.

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 $\mathcal{P}_n$.