Emulating Shared-Memory Do-All Algorithms in Asynchronous Message-Passing Systems*

Dariusz R. Kowalski[2,3], Mariam Momenzadeh[4], and Alexander A. Shvartsman[1,5]

[1] Department of Computer Science and Engineering, University of Connecticut, 371 Fairfield Rd., Unit 1155, Storrs, CT 06269, USA.
[2] Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, Saarbrücken, 66123 Germany.
[3] Instytut Informatyki, Uniwersytet Warszawski, ul. Banacha 2, 02-097, Warszawa, Poland.
[4] Department of Electrical and Computer Engineering, Northeastern University, 409 Dana Research Center, 360 Huntington Ave., Boston, MA 02115, USA.
[5] Computer Science and Artificial Intelligence Laboratory, MIT, 200 Tech Square, Cambridge, MA 02139, USA.

Abstract. A fundamental problem in distributed computing is performing a set of tasks despite failures and delays. Stated abstractly, the problem is to perform N tasks using P failure-prone processors. This paper studies the efficiency of emulating shared-memory task-performing algorithms on asynchronous message-passing processors with quantifiable message latency. Efficiency is measured in terms of work and communication, and the challenge is to obtain subquadratic work and message complexity. While prior solutions assumed synchrony and constant delays, the solutions given here yields subquadratic efficiency with asynchronous processors when the delays and failures is suitably constrained. The solutions replicate shared objects using a quorum system, provided it is not disabled. One algorithm has subquadratic work and communication when the delays and the number of processors, K, owning object replicas, are $O(P^{0.41})$. It tolerates $\lceil \frac{K-1}{2} \rceil$ crashes. It is also shown that there exists an algorithm that has subquadratic work and communication and that tolerates $o(P)$ failures, provided message delays are sublinear.

Keywords: Distributed algorithm, fault-tolerance, work, communication, quorums.

1 Introduction

A fundamental problem in distributed computing is performing N tasks in a distributed system consisting of P processors, and despite the presence of failures. The abstract problem is called Do-All when processors communicate by exchanging messages and

* The work of the first author was performed at the University of Connecticut and supported in part by the NSF-NATO Award 0209588 and by the KBN Grant 4T11C04425; Email: darek@mimuw.edu.pl. This work is based on the second author's Masters thesis [15] at the University of Connecticut; Email: mmomenza@ece.neu.edu. The work of the third author is supported in part by the NSF CAREER Award 9984778 and the NSF Grants 9988304, 0121277, and 0311368; Email: alex@theory.lcs.mit.edu.

M. Papatriantafilou and P. Hunel (Eds.): OPODIS 2003, LNCS 3144, pp. 210–222, 2004.

the tasks are similar in size and independent. Examples of such tasks include searching a collection of data, applying a function to the elements of a matrix, copying a large array, or solving a partial differential equation by applying shifting method. This problem has been studied in different settings, such as the message-passing model [5,6,8], and in the shared-memory model [1,11,12], where the problem is called Write-All. Depending on the model of computation, algorithmic efficiency is evaluated in terms of time, work, and message complexity. Work is defined as either the total number of steps taken by the available processors [11], or the total number of tasks performed [6]. Message complexity is expressed as the total number of point-to-point messages.

It has been observed that maintaining synchrony in real systems is expensive and models incorporating some form of asynchrony are considered to be more realistic. The Do-All problem has been substantially studied for synchronous failure-prone processors [4,6,5,8], however there is a dearth of asynchronous algorithms. This is not that surprising as it was shown by Kowalski and Shvartsman [13]. With the standard assumption that initially all tasks are known to all processors, the problem can be solved by a communication-oblivious algorithm where each processor performs all tasks. Such a solution has work $S = \Theta(N \cdot P)$, and requires no communication. For an algorithm to be interesting, it must be better than the oblivious algorithm, in particular, it must have subquadratic work complexity. However, if messages can be delayed for a "long time" (e.g., $\Theta(N)$ time), then the processors cannot coordinate their activities, leading to an immediate lower bound on work of $\Omega(P \cdot N)$. With this in mind, a delay-sensitive study of Do-All in [13] yields asynchronous algorithms achieving *subquadratic*[1] work as long as the message delay d is $o(N)$. The message complexity is somewhat higher than quadratic in some executions. The question was posed whether it is possible to construct asynchronous algorithms that simultaneously achieve subquadratic work and communication. It appears that in order to do so, one must strengthen the model assumptions, e.g., impose upper bounds on delays and constrain the failure patterns that may occur during the execution.

Contributions. We study emulation of shared-memory task-performing algorithms in asynchronous message-passing systems. Our goal is to obtain subquadratic bounds on work and communication for the Do-All problem:

Given N similar tasks, perform the tasks using P message-passing processors.

We consider an adversary that interferes with the progress of the computation by introducing delays and causing processors to crash. In order to obtain subquadratic work and message complexity, we restrict the power of the adversary. The best previous algorithm that achieves subquadratic work and message complexity [4,9] assumed synchronous processors and constant message latency. In this paper we present the first algorithms for Do-All that meet our efficiency criteria while allowing non-trivial processor asynchrony and message latency.

Let d be the worst case message latency, and e be the worst case time required for a processor to respond to a message (d and e are unknown to the processors). The summary of our results, for $P \leq N$ and a parameter K, is as follows.

[1] That is, the work complexity is $o(P^2 + PN)$, with the usual assumption that $P \leq N$.

1. We show that Do-All can be solved with work $S = O(\max\{K, d, e\}NP^{\log \frac{3}{2}})$ and message complexity $M = O(KNP^{\log \frac{3}{2}})$. The algorithm can be parameterized to have subquadratic work and message complexity of $O(NP^{\delta})$ with $\log \frac{3}{2} < \delta < 1$, when the parameter K is $O(P^{\delta - \log \frac{3}{2}})$ (this is about $O(P^{0.41})$) and when $d, e = O(K)$. The algorithm tolerates $\lceil \frac{K-1}{2} \rceil$ crashes.
2. We show the existence of an algorithm that solves the Do-All problem with work $S = O(\max\{K, d, e\}NP^{\varepsilon})$, and with message complexity $S = O(KNP^{\varepsilon})$, for any $\varepsilon > 0$. The algorithm can be parameterized to have subquadratic work and message complexity of $O(NP^{\delta})$ with $0 < \varepsilon < \delta < 1$, when the parameter K is $O(P^{\delta - \varepsilon})$, and when $d, e = O(K)$. The algorithm tolerates $\lceil \frac{K-1}{2} \rceil$ crashes.
3. We show a lower bound on work of $S = N + \Omega(P \log P)$ for the asynchronous Do-All problem that involves delays, but no processor crashes. (This matches the lower bound [12] for synchronous crash-prone processors, and the lower bound [11] for fail-stop/restartable processors.)

The results (1) and (2) are obtained by analyzing the emulations of shared-memory algorithms X [3] and AWT [1], respectively, in message-passing systems. We use the atomic memory service based on [14]. We replicate certain memory locations needed by the algorithms at K processors for fault-tolerance, with the atomic memory service maintaining replica consistency. The analysis of the algorithms is parameterized in terms of d, e, and K. The adversary is constrained not to disable the quorum configurations used by the atomic memory service, and to respect the bounds d and e.

Related work. The Do-All problem for the message-passing systems was introduced by Dwork, Halpern and Waarts [6]. There is a number of algorithms for the problem in the synchronous message-passing settings [6,5,8,9]. The algorithmic techniques in these papers rely on processor synchrony. Anderson and Woll gave an asynchronous shared-memory algorithm [1] that generalizes the algorithm of Buss et al. [3]. We convert these algorithms to run in message-passing systems with the help of atomic memory emulation. Attiya, Bar-Noy, and Dolev [2] showed how to emulate atomic shared-memory robustly in message-passing systems using processor majorities. Recently Lynch and Shvartsman [14] developed atomic multi-reader/multi-writer memory service for dynamic networks, allowing arbitrary new quorum configuration installations. We use a simplified version of the algorithm [14]. A quorum system is a collection of sets (quorums), where every pair of sets intersect. Quorums can be seen as generalized majorities [10,16], whose intersection property can be used to provide data consistency.

Structure of the document. Section 2 presents the model of computation and a lower bound. In Section 3 we review the shared-memory Write-All algorithms and a shared-memory emulation algorithm. In Sections 4 and 5 we present our message-passing algorithm and its analysis. Conclusions are in Section 6.

2 Model, Definitions, and a Lower Bound

We now define the model of computation, formalize the Do-All problem, define the complexity measures, and present a Do-All lower bound.

Processors. Our distributed system consists of P (asynchronous) processors with unique processor identifiers (PID) from the set $\mathcal{P} = \{0, \ldots, P-1\}$. P is fixed and is known to all processors. Processors communicate by passing messages (discussed later in this section) and have no access to shared memory.

Quorums. We are going to provide a shared-memory abstraction in message-passing systems using *quorum configurations*.

Definition 1 Let $\mathcal{R} = \{R_i\}$ and $\mathcal{W} = \{W_i\}$ be collections of subsets of processor identifiers, such that for all $R_i \in \mathcal{R}$ and for all $W_i, W_j \in \mathcal{W}$, we have $R_i \cap W_j \neq \emptyset$, and $W_j \cap W_i \neq \emptyset$. Then $\mathcal{C} = \langle \mathcal{R}, \mathcal{W} \rangle$ is a *quorum configuration*, where $\mathcal{R}$ is read quorums and $\mathcal{W}$ is write quorums. We use $mem(\mathcal{C})$ to denote the set of processors identifiers appearing in read and write quorums, and $size(\mathcal{C})$ to denote the total number of identifiers, that is, $size(\mathcal{C}) = |mem(\mathcal{C})|$.

Tasks. We consider N tasks, known to all processors. The tasks are similar, i.e., we assume that any task can be performed in one local time step. Task executions do not depend on each other and the tasks are idempotent, i.e., executing a task many times or concurrently has the same effect as executing the task once.

Communication. We assume a message-passing system. The network is fully connected and the processors communicate via *point-to-point* (asynchronous) channels. The delivery is unreliable and unordered, but messages are not corrupted. If a system delivers multiple messages to a processor, it can process these messages in one local time step. Similarly if a processor has several messages to send, the messages can be sent during a single local time step.

Given the motivation in [13] as discussed in the introduction, we are interested in settings where processors are asynchronous, but where there is an upper bound d on message delay (cf. [7]). We also assume that when a processor receives a message requiring a reply, it can send the reply in at most e time units. The bounds d and e need not be constant, and are unknown to the processors — the algorithms must be correct for any d and e.

Adversary Model. A processor's activity is governed by its local clock. We model asynchrony as an adversary that introduces delays between local time steps. The adversary may also cause message loss or delay, and processor crash failures. We use the term *adversary pattern* F to denote the set of events caused by the adversary in a specific execution and the term *adversary model* $\mathbf{F}$ to indicate the set of all adversary patterns for a given adversary.

The adversary is constrained in two ways: (1) the adversary must respect the bounds d and e (defined above), we call it (d, e)-adversary, and (2) when an algorithm uses a quorum configuration $\mathcal{C} = \langle \mathcal{R}, \mathcal{W} \rangle$, the adversary can cause any processor crashes as long as the processors of at least one read quorum and at least one write quorum remain operational and are able to communicate.

Let $\mathcal{Q} = \{\mathcal{C}_i\}$ be the set of all quorum configurations $\mathcal{C}_i$ used by an algorithm. We denote by $\mathbf{F}_{\mathcal{Q}}$ the adversary that respects the above constraints with respect to each $\mathcal{C}_i$.

Remark. Our definition of the adversary is somewhat involved for several reasons. If the adversary is allowed to cause arbitrary message delays, then communication is impossible and work complexity of any algorithm becomes quadratic. Hence we posit an upper bound on message delays. If the adversary is allowed to prevent processors from sending replies expediently, then a similar situation results. Hence we posit an upper bound on the time it takes a processor to send a reply. Lastly, our approach relies on the use of quorum systems. If the adversary is allowed to cause failures that disable the quorum systems used by the algorithms, then shared memory cannot be emulated, again leading to processors acting in isolation and performing quadratic work. Hence we assume that quorum systems are not disabled. *Kramer.*

The Problem. Now we define the Do-All problem.

Definition 2 Do-All: Given P processors and N tasks, perform all tasks for any adversary pattern F in a specific adversary model.

While processors have no access to a global clock, we assume that the time of the local events can be measured on a discrete global clock.

Definition 3 Given a Do-All algorithm for P processors and N tasks, and an adversary model $\mathbf{F}_{\mathcal{Q}}$, the algorithm solves the problem at step τ when all tasks are complete and at least one non-faulty processor has this completion knowledge.

Note that for correctness we do not require that every processor is aware of completion, since the bounds on delays are unknown to the processors.

Complexity Measures. In order to characterize the efficiency of our algorithms, we define two complexity measurements: work and message complexity. We assume that it takes a unit of time for a processor to perform a unit of work according to its local clock, and that a single task corresponds to a unit of work. By this definition the processors are charged for idling or waiting for messages.

Definition 4 Given a problem of size N and a P-processor algorithm that solves the problem at step $\tau(F)$ in the presence of the adversary pattern F in the model $\mathbf{F}_{\mathcal{Q}}$, let $P_t(F)$ denote the number of processors completing a unit of work at global time t, then the work complexity S of the algorithm is: $S_{N,P} = \max_{F \in \mathbf{F}_{\mathcal{Q}}} \{\sum_{t \leq \tau(F)} P_t(F)\}$.

For message-passing settings, message complexity is defined as follows.

Definition 5 Given a problem of size N and a P-processor algorithm that solves the problem at step $\tau(F)$ in the presence of the adversary pattern F in the model $\mathbf{F}_{\mathcal{Q}}$, let $M_t(F)$ be the number of messages sent at global time t, then the message complexity M of the algorithm is: $M_{N,P} = \max_{F \in \mathbf{F}_{\mathcal{Q}}} \{\sum_{t \leq \tau(F)} M_t(F)\}$.

A Lower Bound. A lower bound $\Omega(N + P \log P)$ on work for PRAM Write-All algorithms with crashes was shown by Kedem, Palem, Ragunathan, and Spirakis [12]. The same lower bound was later shown to apply to algorithms in the presence of processor crashes and restarts [3]. These results require the possibility of crashes. Here we show that the same bound holds for the asynchronous setting where no processor fails.

Theorem 1 *Any asynchronous P-processor Do-All algorithm on input of size N has $S_{N,P} = N + \Omega(P \log P)$.*

Proof: We present a strategy for the adversary that results in the worst case behavior. Let A be the best possible algorithm that solves the Do-All problem. The adversary imposes delays as described bellow:
Stage 1: Let $U > 1$ be the number of remaining tasks. Initially $U = N$. The adversary induces no delays as long as the number of remaining tasks, U, is more than P. The work needed to perform $N - P$ tasks when there are no delays is at least $N - P$.
Stage 2: As soon as a processor is about to perform some task $N - P + 1$ making $U \leq P$, the adversary uses the following strategy. For the upcoming iteration, the adversary examines the algorithm to determine how the processors are assigned to the remaining tasks. The adversary then lists the remaining tasks with respect to the number of processors assigned to them. The adversary delays the processors assigned to the first half remaining tasks ($\lfloor \frac{U}{2} \rfloor$) with the least number of processors assigned to them. By an averaging argument, there are no more than $\lceil \frac{P}{2} \rceil$ processors assigned to these $\lfloor \frac{U}{2} \rfloor$ tasks. Hence at least $\lfloor \frac{P}{2} \rfloor$ processors will complete this iteration having performed no more than half of the remaining tasks.

The adversary continues this strategy which results in performing at most half of the remaining tasks at each iteration. Since initially $U = P$ in this stage, the adversary can continue this strategy for at least $\log P$ iterations. Considering these two stages the work performed by the algorithm is: $S_{N,P} \geq \underbrace{N - P}_{Stage\ 1} + \underbrace{\lfloor P/2 \rfloor \log P}_{Stage\ 2} = N + \Omega(P \log P)$. □

In the above strategy the adversary causes at most $\lceil \frac{P}{2} \rceil \log P$ delays, where the processor assigned to the last remaining task is delayed for $\log P$ iterations.

3 Algorithms X, AWT, and Shared Memory Emulation

We now overview two shared-memory algorithms for Write-All, called algorithm X [3] and algorithm AWT [1], and conclude with the shared memory emulation algorithm [14].

Algorithm X. This algorithm has subquadratic work of $S = O(NP^{\log \frac{3}{2}})$ using $P \leq N$ processors. It uses a full binary "progress" tree with P leaves. We assume that the N fixed-size tasks are associated with the leaves, where a "chunk" of $\lceil N/P \rceil$ tasks is positioned at each leaf. The boolean values at the vertices of the progress tree indicate whether or not all work below the current note is complete.

The algorithms proceeds as follows. Acting independently, each processor searches for work in the smallest immediate subtree that has work that needs to be done. The processor performs the work and when all work within the subtree is completed, it marks the root of the subtree as done and moves out of the subtree. The algorithm is presented in detail in [3,11]; its work for any pattern of asynchrony is as follows.

Theorem 2 *[3] Algorithm X solves Write-All of size N with P processors with work $S_{N,P} = O(NP^{\log \frac{3}{2}})$, for $P \leq N$.*

Algorithm AWT. In the algorithm [3], each processor traverses the unvisited subtrees of a vertex within the progress tree according to the permutation $(1, 2)$, i.e., first left, then right, if the bit of its PID at that tree level is 0, and according to the permutation $(2, 1)$, i.e., first right, then left, if the bit of its PID at that tree level is 1. This approach can be generalized to q-ary progress tree algorithms. Here the processors are equipped with q permutations of $\{1, \ldots, q\}$, and each processor traverses the q subtrees at a vertex according to the permutation that corresponds to the q-ary digit of its PID. It is possible to construct a set of q permutations so that the following result holds.

Theorem 3 *[1] Algorithm AWT solves Write-All of size N with P processors with work $S_{N,P} = O(NP^{\varepsilon})$, for any $\varepsilon > 0$ when $P \leq N$.*

Algorithm AWT requires the set of q permutations of $\{1, \ldots, q\}$. The algorithm is correct for any q permutations, however the complexity result holds for a set of q permutations with certain combinatorial properties. These permutations can be found by searching the space of all sets of q permutations, and this space is very large even for moderately small ε. Since we do not show how to construct such permutations, in the rest of the paper we state results depending on algorithm AWT as existential results.

Shared Memory Emulation. We now present the algorithm implementing an atomic memory service based on [14] (the simplified version described here uses a single quorum configuration to access a data object and does not use reconfiguration). We call this atomic memory service *AM*. The algorithm implements read/write shared memory in asynchronous message-passing systems that are prone to message loss and processor crashes. In order to achieve fault-tolerance and availability, *AM* replicates objects at several network locations. In order to maintain memory consistency in the presence of failures, the algorithm uses a *quorum configuration*, consisting of a set of *members* (i.e., the set of processors "owning" a replica) plus sets of *read quorums* and *write quorums*. The quorum intersection property requires that every read-quorum intersect every write-quorum.

Every active node in the system maintains a *tag* and a *value* for each data object. Each new write assigns a unique tag, with ties broken by processor ids. These tags are used to determine an ordering of the write operations, and therefore determine the values that subsequent read operations return.

Read and write operations require two phases, a query phase and a propagation phase, each of which accesses certain quorums of replicas. Assume the operation is initiated at node i. First, in the query phase, node i contacts read quorums to determine the most recent available tag and its associated value. Then, in the propagation phase, node i contacts write quorums. The second phase of a read operation propagates the latest tag discovered in the query phase and its associated value. If the operation is a write operation, node i chooses a new tag, strictly larger than every tag discovered in the query phase. Node i then propagates the new tag, along with the new value, to the write quorums. Note that every operation accesses both read and write quorums. The protocol of each phase is formulated in terms of point-to-point messages. First the messages are sent to the members of the quorum configuration, then the replies are collected until a complete quorum responds.

Each of the two phases of the read or write operations, accesses quorums of processors, incurring a round-trip message delay. Assuming that there is an upper bound d on message delays and that local processing is negligible, this means that each phase can take at most $2d$ time. Given that all operations consist of two phases, each operation takes at most $4d$ time. (The full algorithm that includes a reconfiguration service, and its analysis, is given in [14].)

4 The Message-Passing Algorithm

We now present the emulation of the shared-memory algorithm X in the message-passing model. We call this algorithm *Xmp*. The shared data used in the algorithm is replicated among the processors. Each processor has a local progress tree containing the replicas of the vertices used by the shared-memory algorithm. Specific vertices are *owned* by certain designated processors as described below (each processor has a replica of each vertex, but not all processors are owners).

Memory Management. The progress tree is stored in the array $d[1, \ldots, 2P-1]$. This array is replicated at each processor. We use the index x to denote the vertex $d[x]$ of the progress tree. For each x, let $\mu(x)$ be some non-empty subset of processor identifiers. We call the processors in each $\mu(x)$ the *owners* of the vertex $d[x]$. These processors are responsible for maintaining consistent (atomic) replicas of the vertex. We assume that all *owner* sets in the system have the same size (this is done for simplicity only—both the algorithms and the analysis readily extend to quorum configurations with owner sets of different sizes). For a set Y containing some indices of the progress tree, we let $\mu(Y) = \cup_{x \in Y} \mu(x)$.

When a processor needs to read or write a vertex x in its local progress tree, it uses the atomic memory service *AM*, which accesses the *owners* of the vertex, i.e, processors in $\mu(x)$, until it obtains responses from the necessary quorums (as described in the previous section).

Algorithm Description. Algorithm *Xmp* for each processor has the structure identical to algorithm X, except that each processor has a local copy of the progress tree and may use *AM* to access the vertices of the tree. The processors access the progress tree vertices as follows:

- If a processor needs to read vertex x, and its local value is 0, it uses *AM* to obtain a fresh copy of x.
- If a processor needs to read vertex x, and its local value is 1, the processor uses this value — this is safe because once a progress tree vertex is marked done, it is never unmarked.
- If a processor needs to write (always value 1 according to the algorithm) to vertex x, and its local value is 0, it uses *AM* to write to x and it updates the local value accordingly.
- If a processor needs to write vertex x, and its local value is already 1, the processor does not need to write — once a progress tree vertex is marked done, it is never unmarked.

The tasks are known to all processors and are associated with the leaves of the tree (as in algorithm X). Initially the values stored in each local tree are all zeros. Each processor starts at the leaf of its local tree according to its PID and it continues executing the algorithm until the root of its local tree is set to 1.

Algorithm AWT [1] can also be emulated in the message-passing system considered here using the memory service *AM*. Recall that the main distinction is that algorithm X uses a binary tree, while algorithm AWT uses a q-ary tree.

Correctness. We claim that algorithm *Xmp* (as well as the q-ary tree algorithm based on algorithm AWT) correctly solves the Do-All problem. This follows from the following observations: (1) The correctness of the memory service *AM* (shown in [14]) implies that if a vertex of the progress tree is ever read to be 1, it must be previously set to 1. (2) A processor sets a vertex of the progress tree to 1 if and if only it verifies that its children (if any) are set to 1, or the processor performed the task(s) associated with the vertex when it is a leaf. (3) A processor leaves a subtree if and only if the subtree contains no unfinished work and its root is marked accordingly.

Quorum Configuration Parameterization. We use the size of the *owner* sets, $|\mu(\cdot)|$, to parameterize algorithm *Xmp*. This will allow us later to study the trade-off of algorithm efficiency and fault-tolerance.

Definition 6 For each data object x we define a *quorum configuration* $\mathcal{C}_x = \langle \mathcal{R}_x, \mathcal{W}_x \rangle$, where $mem(\mathcal{C}_x) = \mu(x)$, and $\mathcal{R}_x, \mathcal{W}_x \subseteq 2^{\mu(x)}$. We define K to be the size of the largest quorum configuration, that is, $K = \max_x\{|\mu(x)|\}$. We define $\mathcal{Q}$ to be the set of all quorum configurations, that is $\mathcal{Q} = \{\mathcal{C}_x\}$.

We now discuss the assignment of vertices to owners. Each vertex of the progress tree is owned by K processors. This results in $\binom{P}{K}^{2N-1}$ combinations of "owned" replica placements. Examples 1 and 2 illustrate two possible placements, when $N = P$.

Example 1 Let N and K be powers of 2. The processors are divided into N/K segments with contiguous PIDs. Each segment owns the leaves of the progress tree corresponding to its processors' PIDs along with the vertices in the subtree of the owned leaves. Moreover, each vertex at the height greater than $\log K$ is owned by the owner of its left child. It is not hard to show that the processors with PID $= 0, \ldots, K-1$ (the first segment) own $2K - 1 + \log(N/K)$ vertices but processors with PID $= N-K, \ldots, N-1$ (the last segment) own $2K-1$ vertices. Therefore the first segment processors are more likely to be busier than other processors responding to messages as *owners*. Figure 4 illustrates this example where N=16 and K=4: Here, $\mu(\{1,2,4,8,9,16,17,18,19\}) = \{P_0, P_1, P_2, P_3\}$, $\mu(\{5, 10, 11, 20, 21, 22, 23\}) = \{P_4, P_5, P_6, P_7\}$, etc. (to avoid confusion between tree indices and PIDs we use P_i to denote the processor with PID i).

Example 2 The processors are divided into N/K segments and each segment has K processors with contiguous PIDs. Vertex i of the progress tree is owned by the $j + 1^{th}$ segment, where $i \overset{K}{\equiv} j$. Since there are $2N-1$ vertices in the progress tree, each processor owns either $\lfloor K(2N-1)/N \rfloor$ or $\lceil K(2N-1)/N \rceil$ vertices. Hence there is an almost uniform distribution of vertices among the owners.

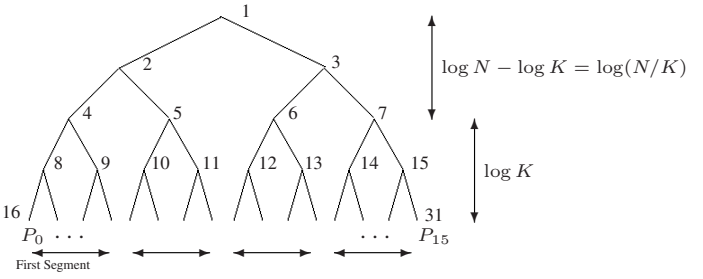

Fig. 1. Owners for $N = 16$ and $K = 4$ (Example 1).

5 Algorithm Analysis

We assess the efficiency of our algorithm against the (d, e)-adversary $\mathbf{F}_{\mathcal{Q}}$ (that respects the delays d and e, and that does not disable the quorum system $\mathcal{Q}$). We start with the following lemma about the cost of reads and writes.

Lemma 4 *Using the atomic memory service AM, each read or write operation contributes at most* $4(K + d) + 2e$ *to the work complexity and at most* $4K$ *to the total number of messages.*

Proof: The processor performing a read or a write operation uses the service *AM* with quorum configurations of size at most K. Thus each quorum consists of no more than K processors. In a single phase (see the overview of the service *AM* in Section 3), it takes no more than K units of work to send messages to K processors. After sending the last message, the processor may have to wait additional $2d + e$ time steps to receive the responses. If some K' responses are received, then it it takes at most K units of work to process them (since $K' \leq K$). Thus it takes time $2K + 2d + e$ to process a single phase. The second phase similarly takes time $2K + 2d + e$. Thus the two phases of an operation take time $4(K + d) + 2e$.

Each stage involves sending K messages and receiving no more than K responses. The total number of messages is no more than $4K$. □

Now we present the work and communication analysis.

Theorem 5 *Algorithm Xmp solves Do-All of size* N *with* $P \leq N$ *processors, for* (d, e)*-adversary* $\mathbf{F}_{\mathcal{Q}}$*, with work* $S_{N,P} = O(\max\{K, d, e\}NP^{\log \frac{3}{2}})$ *and message complexity* $M_{N,P} = O(KNP^{\log \frac{3}{2}})$.

Proof: Using Theorem 2, the work of algorithm X is $S_x = O(NP^{\log \frac{3}{2}})$, counting local operations, and shared-memory read/write operations. In algorithm *Xmp*, each local operation takes one local time step, and each read or write operation takes time $4(K + d) + 2e$ by Lemma 4. Thus the total work is $S_{N,P} = (4(K + d) + 2e) \cdot S_x = (4(K + d) + 2e) \cdot O(NP^{\log \frac{3}{2}}) = O(\max\{K, d, e\}PN^{\log \frac{3}{2}})$.

Using Theorem 2 again, we note that the vertices of the progress tree will be updated by the processors a total of $O(NP^{\log \frac{3}{2}})$ times. In algorithm *Xmp*, by Lemma 4, each update contributes $4K$ messages to the message complexity. Thus the total number of messages is $M_{N,P} = 4K \cdot S_x = O(KNP^{\log \frac{3}{2}})$. □

By using algorithm AWT [1] as the basis, and following the identical emulation approach, we obtain the following result with the help of Theorem 3.

Theorem 6 *There exists an algorithm that solves Do-All of size N with $P \leq N$ processors, for (d, e)-adversary $\mathbf{F}_{\mathcal{Q}}$, with work $S_{N,P} = O(\max\{K, d, e\} \cdot NP^{\varepsilon})$, and with message complexity $M_{N,P} = O(KNP^{\varepsilon})$, for any $\varepsilon > 0$.*

Given K, we use quorum systems with all majorities of size $\lceil (K+1)/2 \rceil$, so that we can tolerate the crash of any minority of processors in a configuration.

Theorem 7 *Algorithm Xmp tolerates any pattern of f failures, for $f \leq \lfloor (K-1)/2 \rfloor$.*

The algorithm is most efficient when $K = 1$, that is when the shared memory has a single owner; unfortunately this is not fault-tolerant at all. Thus we are interested in parameterizations that achieve subquadratic work and communication, while maximizing the fault-tolerance of the algorithm.

Theorem 8 *Algorithm Xmp solves Do-All of size N, with $P \leq N$ processors, and for any adversary pattern in $\mathbf{F}_{\mathcal{Q}}$, with (subquadratic in N and P) work and message complexity of $O(NP^{\delta})$ with $\log \frac{3}{2} < \delta < 1$, when the parameter K is $O(P^{\delta - \log \frac{3}{2}})$ and when $d, e = O(K)$.*

Note that if δ is chosen to be close to $\log 3/2$ (≈ 0.59), the algorithm tolerates only a constant number of failures, but it is as work-efficient as its shared-memory counterpart. As δ approaches 1, the complexity remains subquadratic, while the fault-tolerance of the algorithm improves. In particular, when δ is close to 1, the algorithm is able to tolerate about $O(P^{0.41})$ crashes.

Theorem 9 *There exists an algorithm that solves Do-All of size N, with $P \leq N$ processors, and for any adversary pattern in $\mathbf{F}_{\mathcal{Q}}$, with (subquadratic in N and P) work and message complexity of $O(NP^{\delta})$ with $0 < \varepsilon < \delta < 1$, when the parameter K is $O(P^{\delta - \varepsilon})$, and when $d, e = O(K)$.*

Remark. One may be interested in measuring the impact of using our emulation on the performance (work) of the original shared-memory algorithms. It is difficult to do this directly, given that in message-passing systems there exists a natural interdependence between the efficiency and fault-tolerance in the emulation. In order to lower the cost of the emulation one needs to assume fewer failures. On the other hand, tolerating a linear number of processor failures causes the emulation to impose a linear overhead, resulting in worse-than-quadratic work. This is in contrast with the shared-memory solutions where a large number of failures may not necessarily degrade performance: consider a situation where after the initial failures only a constant number of processors remains—in this case work can be optimally linear in the size of the input. *Kramer.*

6 Conclusion and Future Work

We presented and analyzed emulation of asynchronous shared-memory Do-All algorithms in the message-passing model. We focused on the trade-offs between efficiency and fault-tolerance in our algorithms as we examined how the replication of resources affects efficiency. We also presented a lower bound for the asynchronous Do-All problem that involves delays, but no failures. Several aspects of our work are open for future exploration, not the least of which is the narrowing if the existing gap between the upper and lower bounds. Another direction is to characterize the impact of the emulation efficiency, and the reconfiguration of memory replication, on the upper bounds.

Acknowledgements. The authors thank Lester Lipsky, Alex Russell, and the anonymous referees for suggestions that helped improve the presentation in this paper.

References

1. R.J. Anderson and H. Woll, "Algorithms for the Certified Write-All Problem", *SIAM Journal of Computing*, vol. 26, no. 5, pp. 1277-1283, 1997.
2. H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing Memory Robustly in Message Passing Systems", *Journal of the ACM*, vol. 42, no. 1, pp. 124-142, 1996.
3. J. Buss, P. Kanellakis, P. Ragde, and A. Shvartsman, "Parallel Algorithms with Processor Failures and Delays", *Jour. of Algorithms*, vol. 20, no. 1, pp. 45-86, 1996.
4. B.S. Chlebus, L. Gąsieniec, D. Kowalski, and A.A. Shvartsman, "Bounding work *and* communication in robust cooperative computation", *Proceeding of the 16th Int-l Symp. on Distributed Computing*, Springer LNCS 2508, pp. 295-310, 2002.
5. R. De Prisco, A. Mayer, and M. Yung, "Time-Optimal Message-Efficient Work Performance in the Presents of Faults", *Proceedings of the 13th Symposium on Distributed Computing*, pp. 161-172, 1994.
6. C. Dwork, J. Halpern, and O. Waarts, "Performing Work Efficiency in Presence of Faults", *SIAM Journal on Computing*, vol. 27, no. 5, pp. 1457-1491, 1998.
7. C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony", *Journal of the ACM*, vol. 35, no. 2, pp. 288-323,1988.
8. Z. Galil, A. Mayer, and M.Yung, "Resolving Message Complexity of Byzantine Agreement and Beyond", *Proceedings of the 36th IEEE Symposium on Foundation of Computer Science*, pp. 724-733, 1995.
9. C. Georgiou, D. Kowalski, and A.A. Shvartsman, "Efficient Gossip and Robust Distributed Computation", *Proc. of the 17th International Symposium on Distributed Computing*, pp. 224-238, 2003.
10. D.K. Gifford, "Weighted Voting for Replicated Data", *Proceedings of the 7th Symposium on Operating Systems Principles* , pp. 150-159, 1979.
11. P.C. Kanellakis and A.A. Shvartsman, "Fault-Tolerant Parallel Computation", ISBN 0-7923-9922-6, *Kluwer Academic Publishers*, 1997.
12. Z.M. Kedem, K.V. Palem, A. Raghunathan, and P. Spirakis, "Combing Tentative and Definite Executions for Dependable Parallel Computing", *Proceedings of the 23rd Symposium on Theory of Computing*, pp. 381-390, 1991.
13. D. Kowalski and A.A. Shvartsman, "Performing Work with Asynchronous Processors: Message-Delay-Sensitive Bounds", *Proceedings 22nd ACM Symposium on Distributed Computing*, pp. 211-222, 2003.

14. N. Lynch and A.A. Shvartsman. “RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks”, *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, pp. 173-190, 2002.
15. M. Momenzadeh, *Emulating Shared-Memory Do-All in Asynchronous Message Passing Systems*, Masters thesis, University of Connecticut, 2003.
16. R. Thomas, “A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", *ACM Trans. on Database Sys.*, vol. 4, no. 2, pp. 180-209, 1979.