Cooperative Sharing and Asynchronous Consensus using
Single-Reader Single-Writer Registers

Yonatan Aumann\*\quad Avivit Kapah-Levy\dagger

Abstract

We consider the problem of asynchronous consensus in the shared memory setting with single-reader/single-writer registers. This setting can also be viewed as a message-passing setting where there are no delays or failures of links. We provide a consensus procedure that completes in $O(n \log n \exp(2\sqrt{\ln n \ln(\epsilon \log n \log^* n)}))$ expected total work, for some constant $c$, assuming the value oblivious adversary scheduler. In particular, the total work is less than $O(n^{1+\epsilon})$ for any $\epsilon$. Total work is defined as the total number of operations performed by all processors collectively, including busy-waiting. The solution is based on a new algorithm for a primitive which we call cooperative-sharing. In the cooperative-sharing primitive, initially each processor has a private value and the outcome is that there is at least one input value which becomes known to all processors. We provide a new algorithm for the cooperative sharing in the single-reader/single-writer setting which completes in $O(n \exp(2\sqrt{\ln n \ln(\epsilon \log n \log^* n)}))$ work.

\*Department of Mathematics and Computer Science, Bar-Ilan University, Ramat-Gan, Israel, aumann@cs.biu.ac.il.
\daggerDepartment of Mathematics and Computer Science, Bar-Ilan University, Ramat-Gan, Israel, kaph@cs.biu.ac.il.

1 Introduction.

A Motivating Example. Consider a group of $n$ players who wish to reach consensus on a common value. The players communicate by e-mail, which is assumed to arrive instantaneously (relative to the speeds of the players). The players operate asynchronously, with each player accessing (reading or writing) her e-mail whenever she has time. The operating times are arbitrary and may be determined by an adversary. Thus, each player sends or reads a message whenever she is granted time by the adversary scheduler. How many e-mail messages must be sent in order to reach consensus? We show that an expected $O(n \log n \exp(2\sqrt{\ln n \ln(\epsilon \log n \log^* n)}))$ total number of messages suffice, summed up over all messages sent by all players. In particular, the number of messages is $o(n^{1+\epsilon})$ for any $\epsilon > 0$. The bound holds regardless of which player sent more messages and which sent less, and regardless of when each message was sent (i.e., messages are sent when the players are granted time to send them, not when the algorithm requires the players to do so).

Knowledge Sharing. A fundamental problem in distributed computing is that of sharing information across the network. The problem comes in many variants, but the basic problem can be formulated as follows. Initially, one or more processors hold a private input value. The desired outcome is that all processors know some or all of these input values. Information sharing, in its different forms, is a basic primitive in many distributed tasks, such as consensus, snapshots, and many more. In this paper we focus on the problem in the asynchronous setting. The specific problem we study is formulated as follows. Initially each processor, $P_i$, holds a private value $v_i$. The desired outcome is that there be at least one value $v_{i_0}$, which becomes known to all processors. The processors need not know $i_0$, but only that there exists one such $i_0$. The processors may cooperate in sharing the information. We call this problem the (cooperative) sharing problem.
Sharing and Consensus. The cooperative sharing problem is closely related to the cooperative collect problem, first abstracted by Saks, Shavit and Woll [20], and further studied by Aspnes and Hurwood [6]. In essence, [20] show that if the complexity of collect is \( f(n) \) for some function \( f \), then consensus can be obtained in \( O(f(n) \log n) \) total work. Recently, Aumann [10] proved that cooperative-sharing can be substituted for collect in the consensus construction of [20], assuming the value oblivious adversary (see appendix). Specifically, [10] shows that if sharing can be obtained in \( f(n) \) work, then consensus can be obtained in \( O(f(n) \log n) \) work.\(^1\)

Single-reader/single-writer registers. Using an elegant protocol, Aspnes and Hurwood [6] show how to perform collect in \( O(n \log^3 n) \) work. This readily translates to a sharing procedure with the same complexity. Their protocol is for the shared memory setting assuming multi-reader/single-writer registers. Here, we consider the problem in the single-reader/single-writer setting. In the single-reader/single-writer model each pair of processors communicate via a separate communication line/register.

We consider the single-reader/single-writer model of special interest as it reflects the operation of many real world systems, where communication is only available between pairs of processors. Note that the single-reader/single-writer model can also be viewed as a message passing setting with no delays or failures of links. (Thus, the e-mail example above).

Clearly, the collect and sharing problems become much more difficult in the single-reader setting. In the multi-reader setting a processor can write a value once, and it becomes available to all processors. In single-reader setting, in contrast, a processor must communicate the value separately to each other processor. Thus, the results for multi-reader registers do not extend to the single-reader model.

Our Results. In this paper we give an efficient protocol for cooperative sharing in the single-reader/single-writer model. We prove:

**Theorem 1.1.** Assuming: an asynchronous shared memory system; single-writer/single-reader, atomic read/write registers; a value oblivious adversary scheduler. There exists a constant \( c \) such that cooperative sharing can be obtained in expected \( O(n \exp(2 \sqrt{\ln n \ln(c \log n \log^2 n)}) \) total number of shared memory accesses.

The value oblivious adversary is the adversary scheduler as defined in [11]. This is a fully dynamic adversary which knows all the history of operations in the system but is ignorant of the actual values stored in memory, insofar as they do not affect the actions of the processors. A more detailed description of the adversary model is provided in the appendix.

As a corollary, we obtain:

**Theorem 1.2.** Assuming: an asynchronous shared memory system; single-writer/single-reader, atomic read/write registers; a value oblivious adversary scheduler. There exists a constant \( c \) such that asynchronous consensus can be obtained in expected \( O(n \log n \exp(2 \sqrt{\ln n \ln(c \log n \log^2 n)}) \) total number of shared memory accesses.

This is, to the best of our knowledge, the first algorithm for the single-reader model to obtain consensus in less than \( O(n^2) \) work.

We note that the value oblivious adversary assumption is necessary. For the strong adversary scheduler Aspnes [4] proved that \( \Omega(n^2) \) work is necessary even in the multi-reader setting. Thus, to cross the \( \Omega(n^2) \) barrier a weakened adversary model is essential.

\(^1\)In the original paper [10], the claim was stated that collect with collective freshness suffices. However, the actual proof provides that sharing (which was not yet defined at the time) is sufficient.
Work is measured by the total number of steps performed in the system, summed up over all processors. Following [20, 6] and others, we count only shared memory accesses (reads and writes). We assume poly-$n$ size registers.

Outline. The rest of this paper is organized as follows. Immediately hereunder we provide an overview of related work. In Section 2 we provide the necessary definitions and notations. The main result in presented in two stages. First, in Section 3 we give a protocol for sharing which completes in $O(n^{3/2} \log^2 n)$ work. Then, in Section 4 we provide a recursive construction, which, based on the $O(n^{3/2})$ protocol, gives the final result.

1.1 Related Work. The collect problem was first abstracted and applied to consensus by Saks, Shavit and Woll [20]. A theory of competitive analysis for the distributed computing, together with competitive implementations of collect are presented by Ajtai, Aspnes, Dwork and Waarts [2] and by Aspnes and Waarts [8]. Aspnes and Hurwood [6] provide efficient solutions for various asynchronous collect and rumor spreading problems, in the multi-reader setting. In particular, they give an elegant $O(n \log^3 n)$ implementation of collect, and an almost tight $\Theta(\sqrt{n})$ bound on the throughput competitiveness of fresh-collect. For the synchronous setting, the well known gossip problem is the analogue to the asynchronous collect. For efficient algorithms for gossip see [16, 17].

Asynchronous consensus is discussed in a multitude of papers. Fischer, Lynch, and Paterson [18] prove the impossibility of deterministic asynchronous consensus in the message-passing model, even if only one processor fails. Chor, Israeli, and Li [14] and Loui and Abu-Amara [19] show that the same holds in the shared-memory model (also see [15]). Randomized solutions for the full knowledge adversary scheduler are in [1, 5, 9, 3, 12, 7] and others. The best of these solutions require $\Omega(n)$ work per-processer, for a total of $\Omega(n^2)$ work.

Recently, Aspnes [4] proved that for the strong adversary scheduler, $\Omega(n^2)$ work is necessary. Thus, if we seek to break this bound a different adversary model must be used. A weaker adversary model was already considered in the original papers of Chor, Israeli and Li [14] and Abrahamson [1]. However, their results do not improve upon $O(n^2)$ total work bound. Recently, the problem of improving the consensus bounds using weaker, but still realistic, adversary models was re-considered in [11, 13, 10]. Aumann and Bender [11] and Chandra [13] define "intermediate" adversary models, and provide $O(n \log^2 n)$ consensus algorithms, using multi-reader/multi-writer registers ([13] also provides an $O(\log^2 n)$ bound per-processer). Aumann [10] provides an $O(n \log^4 n)$ solution using single-reader/multi-writer registers. In this paper we consider the single-reader/single-writer model.

2 Preliminaries.

2.1 Definitions and Notations. There are $n$ processors in the system, $P_1, \ldots, P_n$. At any given time, for any given processor, $P_i$, there is some information "known" to the processor and some which is not known. We view the "knowledge" of $P_i$ as a set of pairs ("var-name", value) where "var-name" is a symbolic name of a variable and value is its value. Formally, for each processor $P_i$ and time $t$ we define the knowledge-set of $P_i$ at time $t$, denoted by $KS_t(i)$, $KS_t(i) = \{(\text{"var}_1, \text{val}_1), \ldots, (\text{"var}_m, \text{val}_m)\}$. We omit the time designation and write $KS(i)$ to denote the current knowledge-set of $P_i$.

Using these notations the sharing problem can be formulated as follows. Initially, the knowledge-set of each processor $P_i$ consists of a single element: $(\text{"v}_i, \text{v}_i)$. We call $v_i$ the private input of $P_i$. For each processor $P_i$, let $t_i$ be the time $P_i$ completes its part in the sharing protocol. The desired outcome of the sharing protocol is that there exist at least one $t$ such that for each $i$ ($i = 1, \ldots, n$), $(\text{"v}_{t_0}, \text{v}_{t_0}) \in KS_{t_0}(i)$. Note that for any processor $P_i$, the knowledge set $KS_{t_i}(i)$ may contain many
input values $v_j$. In this case the processor need not know which value is actually shared by all processor. The only requirement is that there exist one such $v_{in}$.

The processors communicate using shared registers/buffers. For each pair of processors $P_i$ and $P_j$ there is a buffer $\text{Buffer}(i, j)$ to which $P_i$ can write, and from which $P_j$ can read. No other processor can write to, and no other processor can read from this buffer. We denote by $\text{Buffer}_{t_i}(i, j)$ the content of the buffer at time $t$. The protocol runs in steps. At each step exactly one processor is active. The active processor, $P_t$, can perform one of two actions:

1. **Write**: Chose a $j$ and write any part of the current knowledge set $KS(i)$ to $\text{Buffer}(i, j)$.

2. **Read**: Choose a buffer $\text{Buffer}(j, i)$ and read its content. In this case, the entire content of the buffer is added to the knowledge set of $P_t$, i.e. $KS(i) = KS(i) \cup \text{Buffer}(j, i)$. (Thus we count only communication steps, not internal computations.)

At each step the processor to operate is determined by an adversary scheduler. The adversary scheduler is the value-oblivious-adversary scheduler, as defined in the appendix.

### 2.2 Scatter and Gather

For the sharing algorithm we use two sub-procedures: **scatter** and **gather**. Intuitively, a scatter procedure is a procedure by which one set of processors distribute their knowledge to another set. Specifically, for sets of processors, $S$ and $T$, called the *source set* and *target set*, respectively, and set of variables $V$, an $(S,T)$-scatter of $V$ is a procedure by which the processors in $S$ distribute their knowledge of the values of the variables in $V$ to a all processors of $T$. Note, however, that we cannot require that the processors of the target set actually know the information, as they may not be operating and thus, never get to read the information written to them. Thus, we only require that the information be resident at the buffers of $T$. Formally,

**Definition 1.** For a knowledge set $KS(i)$ and set of variable names $V$, denote by $KS^V(i)$ the variable in $V$ which are in $KS(i)$, i.e. $KS^V(i) = \{ (v, v) \in KS(i) | v \in V \}$.

Let $S$ and $T$ be sets of processor indexes, and $V$ a set of variable names. An $(S,T)$-scatter of $V$ is a procedure for the processors of $S$ such that the following holds. For any $i \in S$, let $t_i$ be the time when the $P_i$ starts its part in the procedure, and let $t_{end}$ be the time when the first processor of $S$ completes its part in the procedure. Then, for any $j \in T$

$$\bigcap_{i \in S} KS^V_i(i) \subseteq \bigcup_{i \in S} \text{Buffer}_{t_{end}}(i, j).$$

That is, any knowledge on $V$ known to all members of $S$ prior to the beginning of the procedure must, by the end of the procedure, be available to any member of $T$, if it only were to read the information in the buffers written by $S$. Note that the definition refers to the time when the first processor of $S$ completes its part in the procedure, which is not necessarily when all processors of $S$ complete. This is because we can never wait for all processors to end, as some may be faulty or slow.

Gather is the complement procedure of scatter. Intuitively, an $(S,T)$-gather is the procedure by which the processors of a target set $T$ collectively read the information previously written to them by a set $S$ (here we do not restrict the gather to a specific set of variables, though the definition could easily extend to this setting). Formally,

**Definition 2.** Let $S$ and $T$ be a sets of processor indexes. An $(S,T)$-gather is a procedure for the processors of $T$ such that the following holds. Let $t_0$ be the time when the procedure starts, and for
any \( j \in T \) let \( t_j \) be the time when \( P_j \) completes its part in the procedure. Then, for all \( j \in T \),

\[
\text{KS}_{t_j}(j) \subseteq \bigcap_{j \in T} \bigcup_{i \in S} \text{Buffer}_{t_0}(i, j).
\]

That is, any knowledge initially available to all members of \( T \) in the buffers written by \( S \), is, at the end of the procedure actually incorporated into their knowledge sets.\(^2\)

We denote by an \( \alpha \)-scatter an \((S,T)\)-scatter with \(|T| = |S|^{\alpha} \). Similarly, an \( \alpha \)-gather is an \((S,T)\)-gather with \(|S| = |T|^{\alpha} \).

3 An \( O(n^{3/2} \log^2 n) \) Cooperative-Sharing Protocol.

3.1 General Structure. For the \( O(n^{3/2} \log^2 n) \) protocol the \( n \) registers are divided into \( \sqrt{n} \) sub-sets, \( S_1, \ldots, S_{\sqrt{n}} \). The algorithm works in three phases (later we explain how each phase is achieved):

**Phase 1:** Each sub-set \( S_j \) separately performs cooperative-sharing on its input values, \( \{ "v_i" : i \in S_j \} \) ("\( v_i \)" is the private input value of \( P_i \)).

**Phase 2:** Each sub-set \( S_j \) (separately) performs an \((S_j, [1, n])\)-scatter of \( \{ "v_i" : i \in S_j \} \), where \([1, n]\) is the set of all processors.

**Phase 3:** Each sub-set \( S_j \) (separately) performs an \(([1, n], S_j)\)-gather (in which all input values are gathered).

In all of the phases processors never wait for each other to complete their respective parts in the scatter, gather and sharing procedures. Rather, each processor continues to the next phase immediately after it completes its part in the phase.

The rest of this section is devoted to explaining how each phase is carried out. Before doing so we first prove that the protocol guarantees sharing, as defined in Section 2.

**Claim 1.** Upon completion of Phases 1-3, sharing is obtained.

**Proof.** Let \( S_{j_0} \) be the first sub-set to have a processor complete Phase 2, and let \( t_2 \) be the time this happens. Let \( v_{i_0} \) be the value shared by \( S_{j_0} \) is Phase 1. By definition, any processor \( P_i \) of \( S_{j_0} \) completing Phase 1 has \( \{ "v_{i_0}" , v_{i_0} \} \in \text{KS}(i) \). Thus, any processor \( P_i \) of \( S_{j_0} \) “knows” the value of \( v_{i_0} \) by the time it starts Phase 2. Thus, by the definition of scatter, by \( t_2 \), for any other sub-set \( S_{j'} \) and any processor \( i \in S_{j'} \) \( \{ "v_{i_0}" , v_{i_0} \} \in \text{Buffer}_{t_0}(i', i) \). Thus, the value of \( v_{i_0} \) is gathered by all processors in Phase 3.

We now proceed to describe the operation of each phase separately.

3.2 Phase 1. The procedure for Phase 1 is simple. First, each processor \( P_i \), \( i \in S_j \), writes its value to all members of \( S_j \), one by one in sequence. Then, the processor reads the information written to it by all other processors of \( S_j \), one by one in sequence. The work per-processor is \( 2 \sqrt{n} \), for a total of \( 2n^{3/2} \) for all processors.

\(^2\)We note that the gather procedure is only necessary since, in the shared memory setting, a processor does not know when other processors write to its buffers. If there is some indication when and where the writes occur [e.g. phone calls] then the gather procedure is superfluous.
\[(S, T)\text{-scatter} \text{ of variables } V \text{ for processor } P_i\]

1. repeat
2. \textbf{forall} \( j \in S \) do
3. \quad read \text{Buffer}(j, i) \quad \{ \text{get information from other sub-sources} \}
4. \quad \text{KS}(i) = \text{KS}(i) \cup \text{Buffer}(j, i)
5. \quad R = \{ k : (\text{"w}_k", 1) \notin \text{KS}(i) \} \quad \{ \text{sub-targets not yet written} \}
6. \quad \text{choose} \ x \in R \text{ at random}
7. \quad \textbf{forall} \ j \in T \text{ do}
8. \quad \text{write} \ \text{Buffer}(i, j) \leftarrow \text{KS}^V(i) \quad \{ \text{write values to all members of sub-target} \}
9. \quad \text{KS}(i) = \text{KS}(i) \cup \{(\text{"w}_x", 1)\} \quad \{ \text{update completed targets list} \}
10. \quad \textbf{forall} \ j \in S \text{ do}
11. \quad \text{write} \ \text{Buffer}(i, j) \leftarrow \text{KS}^W(i) \quad \{ W = \{\text{"w}_1", \ldots, \text{"w}_p\} \}
12. \text{until} \ R = \emptyset

\text{Figure 1: 2-scatter}

\[(S, T)\text{-gather} \text{ for processor } P_i\]

1. repeat
2. \textbf{forall} \( j \in T \) do
3. \quad read \text{Buffer}(j, i) \quad \{ \text{obtain information from other members of } T \}
4. \quad \text{KS}(i) = \text{KS}(i) \cup \text{Buffer}(j, i)
5. \quad R = \{ k : (\text{"w}_k", 1) \notin \text{KS}(i) \} \quad \{ \text{sub-sources not yet read} \}
6. \quad \text{choose} \ x \in R \text{ at random}
7. \quad \textbf{forall} \ j \in S \text{ do}
8. \quad \text{read} \ \text{Buffer}(j, i) \quad \{ \text{read buffers written by sub-source} \}
9. \quad \text{KS}(i) = \text{KS}(i) \cup \text{Buffer}(j, i) \quad \{ \text{update knowledge set} \}
10. \quad \text{KS}(i) = \text{KS}(i) \cup \{(\text{"w}_x", 1)\} \quad \{ \text{communicate knowledge set to the members of } S \}
11. \quad \textbf{forall} \ j \in S \text{ do}
12. \quad \text{write} \ \text{Buffer}(i, j) \leftarrow \text{KS}(i) \quad \{ U = \{\text{"u}_1", \ldots, \text{"u}_m\} \}
13. \text{until} \ R = \emptyset

\text{Figure 2: 2-gather}

### 3.3 Phase 2 - A 2-scatter protocol.

Phase 2 requires an \((S_j, [1, n])\)-scatter, where \(|S_j| = \sqrt{n}\) and \([1, n] = n \). Since \([1, n] = |S_j|^2\), what we need a 2-scatter. We give a general construction for a 2-scatter.

Consider a source set \(S\) and a target set \(T\), set \(p = T\) (we use \(p\) rather than \(n\) to emphasise the general nature of the construction). The target set \(T\) is divided into \(\sqrt{p}\) sub-targets, \(T_1, \ldots, T_{\sqrt{p}}\), each of size \(\sqrt{p}\). Each sub-target is going to be written as one unit, sequentially, by the same processor. Processors of \(S\) choose the sub-target to which they write randomly. To avoid repetition, processors of \(S\) keep track of which sub-targets have already been written, and choose only from the remaining sub-targets. To this end, we define \(\sqrt{p}\) boolean variables \(w_1, \ldots, w_{\sqrt{p}}\). Initially, \(w_x = 0\) for all \(x\). Setting \(w_x = 1\) represents the fact that sub-target \(T_x\) was fully written. Accordingly, whenever a processor \(P_i\) finishes writing a sub-target \(T_x\), it adds \(\text{("w}_x", 1)\) to its knowledge-set. It then communicates its knowledge-set to all other members of \(S\). By doing so, \(P_i\) communicates to the other processors the information that \(T_x\) has been written, as well as any information it might have previously obtained from other processors. A processor completes the procedure when it knows that all sub-targets have been written. A full description of the procedure appears in Figure 1.

6
Proof. Consider a stage $\Pi_\ell = [t_\ell, t_{\ell+1})$. Suppose that $|K_{t_\ell}| = \alpha m$. Let $x \in K_{t_\ell}$, Then,

$$\Pr [x \in K_{t_{\ell+1}}] \leq e^{-d/\alpha}.$$  

Proof. Consider $x \in K_{t_\ell}$. If there exists a complete round, $C$, in $\Pi_\ell$ for which either: (i) $x \notin R$ (line 5), or (ii) $x$ is chosen in line 6, then $x$ will not be in $K_{t_{\ell+1}}$. This is because $C$ knows $x$ was written, and it ends the round by communicating this knowledge to all other members of $S$ (lines 10-11). Thus, for $x \in K_{t_{\ell+1}}$, it must be that in all complete rounds of $\Pi_\ell$, $x \in R$ in line 5, and it is not chosen in line 6. A round starts with reading all input buffers. Thus, for any round, $|R| \leq |K_{t_\ell}| \leq \alpha m$. There are at least $dm$ complete rounds in $\Pi_\ell$. Thus,

$$\Pr [x \in K_{t_{\ell+1}}] \leq \left(1 - \frac{1}{\alpha m}\right)^{dm} \leq e^{-d/\alpha}.$$  

Thus,

**Corollary 1.** Consider a stage $\Pi_\ell = [t_\ell, t_{\ell+1})$. Suppose that $|K_{t_\ell}| = \alpha m$. Then,

$$\Pr \left[|K_{t_{\ell+1}}| > 2\alpha e^{-d/\alpha} m \right] \leq \frac{1}{2}.$$  

Proof. Linearity of expectations and Markov inequality.

**Corollary 2.** The expected number of stages until $K_{t} = \emptyset$ is $O(\log^* m)$.

**Corollary 3.** The 2-scatter on $|T| = p$ completes in expected $O(p\log^* p)$ work.

Proof. Each round takes $O(m)$ work. Each stage consists of at most $(d + 1)m$ rounds. After an expected of $\log^* m$ stages $K_t = \emptyset$. Once $K_t = \emptyset$ each processor can perform at most one round, before it knows that all sub-targets have been written. Thus, the total expected work is bounded by $(d + 1)\log^* m + 1) m \cdot m = O(m^2 \log^* m) = O(p\log^* p)$. 

3.4 Phase 3 - 2-gather. The protocol for 2-gather is analogous to the scatter algorithms. Consider an \((S,T)\)-gather, with \(|S| = p\) and \(|T| = \sqrt{p}\). We call \(S\) the source and \(T\) the target. We divide the source set into \(\sqrt{p}\) sub-sources each of size \(\sqrt{p}\). A processor \(P_i\) of \(T\), chooses a sub-source \(S_x\) which has not yet been read, at random, and reads all the buffers written by the processors of the sub-source \(S_x\) (naturally, \(P_i\) only reads those buffers which it can read, i.e., those written to it). It then incorporates this knowledge to its own knowledge set, and then communicates the knowledge to all other members of \(T\). In addition, \(P_i\) also informs the other members of \(T\) of the sub-sources it \((P_i)\) knows that have been read. A processor completes its part in the procedure when it knows that all sub-sources have been read. The full description appears in Figure 2. The \(S_x\)'s denote the sub-sources, and \(("u_{x}\", 1)\) represents the fact that the information from the sub-source \(S_x\) has been read.

3.5 Complexity. Phase 1 takes \(O(n^{3/2})\) work each. Phases 2 takes \(O(p\log^* p)\) per \(S_j\), where \(p = n\). There are \(\sqrt{n}\) sources \(S_j\). Thus, the total work for Phase 2 is \(O(n^{3/2}\log^* n)\). Similarly for Phase 3. Thus, we obtain,

Claim 4. Cooperative sharing can be completed in expected \(O(n^{3/2}\log^* n)\) work.

4 The Recursive Construction.

4.1 General Structure. Building upon the results of the previous section, we give a recursive construction, which for any \(i \leq \log n\) provides cooperative sharing in \(n^{i+2} (c \log n \log^* n)^i\), for some constant \(c\). Plugging-in the optimal value for \(i\) we obtain the result of Theorem 1.

The inductive construction works as follows. For a given \(i\), we divide the \(n\) processors into \(n^{1/r}\) sub-sets, \(S_1, \ldots, S_{n^{1/(i+1)}}\), each of size \(n^{1/r}\). Each sub-set \(S_j\) performs the following three phases:

**Phase 1:** Cooperative-sharing of the input values \(V_j = \{"u_i": i \in S_j\}\) within the sub-set.

**Phase 2:** \((S_j, [1, n])\)-scatter of \(V_j\) (where \([1, n]\) is the set of all processors).

**Phase 3:** \(([1, n], S_j)\)-gather (of all input values).

Note that the algorithm of the Section 3 is a special case of the construction with \(i = 1\).

The main issue is, of course, how to perform each of the Phases 1-3. For a given \(i\), for any sub-source \(S_j\) we have \(\lceil 1, n \rceil = \lceil S_j \rceil^{i+1}\). Thus, we need:

1. A cooperative sharing procedure (Phase 1),
2. An \(\lceil i+1 \rceil\)-scatter procedure (Phase 2), and
3. An \(\lceil i+1 \rceil\)-gather procedure (Phase 3).

Each of the above three procedures, sharing, scatter and gather, will be implemented using a recursive call to the same procedures for smaller \(i\)'s.

4.2 The Inductive Claims. By induction on \(i\), assume that for any \(p\), the size of the target set, we have for some constant \(c\):

1. A cooperative-sharing procedure which completes in \(p^{i+2} (c \log p \log^* p)^i\) work.
2. An \(\lceil i+1 \rceil\)-scatter procedure which completes in \(p (c \log p \log^* p)^i\) work.
3. An \( \frac{i+1}{i+1} \)-gather procedure which completes in \( p (c \log p \log^* p)^i \) work.

The construction of Section 3 provides the basis of the inductive hypothesis, for \( i = 1 \). (The results of Section 3 are stronger, since they do not have the \( \log p \) factor. However, this makes little difference in the recursive structure, but would make the equations more complex).

Given these procedures we construct:

1. A cooperative-sharing procedure which completes in \( p^{\frac{i+1}{i+2}} (c \log p \log^* p)^{i+1} \) work.
2. An \( \frac{i+2}{i+1} \)-scatter protocol which completes in \( p (c \log p \log^* p)^{i+1} \) work.
3. An \( \frac{i+2}{i+1} \)-gather protocol which completes in \( p (c \log p \log^* p)^{i+1} \) work.

4.3 The \( \frac{i+2}{i+1} \)-scatter Procedure.

The Algorithm. We provide a construction for a general source \( S \) and target \( T \), with \( |T| = p \) and \( |S| = p^{\frac{i-1}{i+2}} \). For the \((S,T)\)-scatter procedure we divide \( S \) into \( p^{\frac{1}{i+2}} \) sub-sources, \( S_1, \ldots, S_{p^{1/(i+2)}} \) each of size \( p^{\frac{i-1}{i+2}} \). Similarly, we divide \( T \) into \( p^{\frac{1}{i+2}} \) sub-targets, \( T_1, \ldots, T_{p^{1/(i+2)}} \), each of size \( p^{\frac{i-1}{i+2}} \).

We now perform a procedure similar to the one of the Section 3, with processors choosing a sub-target at random, and writing their values to all processors of the chosen sub-target. In the recursive construction, however, an entire sub-source acts as one unit: collectively choosing the sub-target, and collectively writing the values to it. The collective choice is attained using a consensus procedure. The collective write is a recursive application of the scatter procedure. The key observation is that for any sub-source \( S_j \) and sub-target \( T_x \), \( |T_x| = |S_j|^\frac{i+2}{i+1} \). Thus, the scatter is an \( \frac{i+1}{i+2} \)-scatter, which is exactly the scatter already available from previous level of the inductive construction.

In addition to choosing the sub-target \( T_x \) and writing to it, the sub-source must also collectively inform the other processors (of \( S \)) that the \( T_x \) was chosen and successfully written, and collectively gather whatever information is provided by the other processors. This is performed by another application of scatter and gather. Again, the key observation is that for any sub-source \( S_j \), \( |S| = |S_j|^\frac{i+1}{i+2} \). Thus, we can again apply the scatter and gather from the previous level of the inductive construction.

\begin{verbatim}
Scatter_{\frac{i+2}{i+1}} (S, T, V) for processor \( P_i \) of sub-source \( S_j \):

1 repeat
2    participate in Gather_{\frac{i+1}{i}} (S, S_j) \{ gather information from other sub-sources \}
3    \( R = \{ k : ( \text{"w}_k \" , 1) \not\in KS(i) \} \)
4    choose \( x_i \in R \) at random
5    Run consensus procedure with all members of \( S_j \) agree on \( x \) (use \( x_i \) as private input value).
6    participate in Scatter_{\frac{i+2}{i+1}} (S_j, T_x, V) \{ collectively write to sub-target \}
7    KS(i) = KS(i) \cup \{ ("w_k", 1) \} \{ update knowledge set \}
8    participate in Scatter_{\frac{i+1}{i+2}} (S_j, S, W) \{ inform the other sub-sources of the write \}
9 until \( R = \emptyset \)
\end{verbatim}

Figure 3: \( \frac{i+2}{i+1} \)-scatter

A detailed description of the procedure is provided in Figure 3. We use the following notations. For a source \( A \) and target \( B \) (with \( |B| = |A|^\frac{i+1}{i+2} \)), and set of variable \( U \), we denote by
Gather\(_{i+1}(S,T)\) for processor \(P_i\) of sub-target \(T_j\)

1 repeat
2 Gather\(_{i+1}(T,T_j)\) \{ gather information from other sub-target \}
3 \(R = \{ k : ("u_k", 1) \notin KS(i) \}\) \{ sub-sources not yet read \}
4 choose \(x_i \in R\) at random
5 Run consensus procedure with all members of \(T_j\) to agree on \(x\) (use \(x_i\) as private input value).
6 Gather\(_{i+1}(S_x,T_j)\) \{ collectively read information from to sub-source \}
7 KS\(_i\) = KS\(_i\) \(\cup \{("u_x",1)\}\) \{ update knowledge set \}
8 Scatter\(_{i+1}(S_j,S,all)\) \{ inform the other sub-targets of the read \}
9 until \(R = \emptyset\)

Figure 4: \(i+2\)th gather

Scatter\(_{(i+1)/i}(A,B,U)\), the \(i+1\)th scatter procedure available by the inductive hypothesis. Similarly, Gather\(_{(i+1)/i}(A,B)\), is the \(i+1\)th gather procedure available by the inductive hypothesis. In Figure 3, the variables \(V\) denote the set of variables to be scattered. The variables \(W = \{"w_1", \ldots,\}\) are boolean variables indicating the successful writing to the sub-targets \(T_x\). Throughout, it is assumed that all scatter, gather and consensus procedures are numbered so that processors can identify the procedures in which they are participating.

**Consensus.** As mentioned above, the processors of a sub-source \(S_j\) collectively choose a random sub-target to write by employing a consensus procedure. We use a consensus procedure based on [10]. The [10] construction provides that if sharing is performed in \(f(p)\) work, for some function \(f\), then consensus is obtained in \(O(f(p) \log p)\). Thus, we employ the sharing procedure available from the previous level of the inductive construction. One difference between the [10] result and the one we need here is that the [10] result, as stated, provides consensus on a single bit, while we need a collective choice of a sub-target. However, the actual [10] construction provides an ID-consensus procedure, where the processors can agree on the ID-number of one of the active processors, thus collectively electing one processor. Furthermore, the construction provides that the elected processor can broadcast information to all other processors, by use of the consensus protocol itself. Thus, in our case, prior to starting the consensus procedure each processor chooses a sub-target at random. Then, the processors perform the consensus procedure, and the elected processor broadcasts his choice to all other processors. The sub-target chosen by the elected processor is used as the sub-target for the entire sub-source. Since we assume a value-oblivious adversary, the choice of sub-target is unknown to the scheduler until after some processor starts acting upon the choice, which is after the consensus has been reached. Thus, the adversary cannot influence the choice. A detailed description of the changes used here, with regards to [10], and the related proofs, will be provided in the full version of paper.

**Analysis.** The analysis is essentially identical to that of the previous section. The only difference is that we must now substitute entire sub-sources for single processors, and plug-in the complexity of the gather, scatter and sharing procedures from the inductive hypothesis. We sketch the proof.

We define the notion of a **rounds** for sub-sources. Intuitively, a round is the time when a sub-source collectively performs lines 2-8. Following is a formal definition. For a sub-source \(S_j\), round 1 starts when the first processor of \(S_j\) starts line 2, and ends the first time a processor of \(S_j\) ends line 8. Assume round \(k\) was defined. Round \(k + 1\) starts at the first time after the end of round
Given this definition of rounds, we can now apply the exact same analysis as in the Section 3. Specifically, we divide the operation of the procedure into stages, each containing \((d+1)p^{\frac{i}{i+1}}\) starting points of rounds. By Corollary 2, with \(m\) sub-sources and sub-targets after \(c_1 \log^k m = O(\log^k m)\) stages the scatter is complete. Here, \(m = p^{\frac{i}{i+2}}\). Thus, after \(c_1 \log^k \left(p^{\frac{i}{i+2}}\right)\) stages the scatter is complete. Thus, after \(c_1 \log^k \left(p^{\frac{i}{i+2}}\right)(d+1)p^{\frac{i}{i+2}}\) rounds, the scatter is completed.

It remains to determine the complexity of a round. The work of a round is composed of:

1. Gather\(\frac{i+1}{i+2}(S, S_j)\) in line 2. Here, \(|S| = p^{\frac{i+1}{i+2}}\). Thus, by the inductive hypothesis, the work for Gather\(\frac{i+1}{i+2}(S, S_j)\) is

\[
P^{\frac{i+1}{i+2}} \left(c \log p^{\frac{i+2}{i+1}} \log^k \left(p^{\frac{i+1}{i+2}}\right)\right)^i \leq P^{\frac{i+1}{i+2}} \left(c \log p \log^k p\right)^i.
\]

2. Scatter\(\frac{i+1}{i+1}(S_j, T_x, V)\) of line 6 and Scatter\(\frac{i+1}{i+1}(S_j, S, W)\) of line 8. In both cases, the target set is of size \(p^{\frac{i+1}{i+2}}\). Thus, again by the inductive hypothesis, the work for each scatter is:

\[
P^{\frac{i+1}{i+2}} \left(c \log p \log^k p\right)^i.
\]

3. The consensus procedure of line 5. For \(m\) processors, consensus takes \(c_2 \log m = O(\log m)\) times the complexity of the cooperative sharing. Here, \(m = |S_j| = p^{\frac{i+1}{i+2}}\). Together with the inductive hypothesis we get that the work for the consensus is:

\[
c_2 \left(p^{\frac{i+2}{i+1}}\right)^{\frac{i+2}{i+1}} \left(c \log p^{\frac{i+2}{i+1}} \log^k p^{\frac{i+1}{i+2}}\right)^i \log(p^{\frac{i+1}{i+2}}) \leq c_2 p^{\frac{i+1}{i+2}} \left(c \log p \log^k p\right)^i \log p.
\]

In all, the total work per round is:

\[
\leq (2 + c_2) p^{\frac{i+1}{i+2}} \left(c \log p \log^k p\right)^i \log p.
\]

There are at most \(c_1 \log^k p^{\frac{1}{i+2}}(d+1)p^{\frac{1}{i+2}}\) rounds before the scatter is complete. Thus, the total work for the scatter is:

\[
\leq (d+1)c_1 (2 + c_2) p^{\frac{i+1}{i+2}} \left(c \log p \log^k p\right)^i \log p \log^k p
\leq p^{c'} \left(c \log p \log^k p\right)^{i+1},
\]

for \(c' = (d+1)c_1 (2 + c_2)/c\). (note that \(c'\) can be made as small as necessary, with a larger choice of \(c\)).

4.4 The \(\frac{i+2}{i+1}\)-gather Procedure. The protocol for gather is analogous to the scatter algorithms. The full description appears in Figure 4. We use \(S_x\) to denote the sub-sources, and \(\langle \text{"}_x\rangle\),1\) represents the fact that the information from the sub-source \(S_x\) has been gathered.
4.5 Cooperative-Sharing. Using the $\frac{i+2}{i+1}$-scatter and $\frac{i+2}{i+1}$-gather procedures, we can now construct the cooperative sharing procedure. In level $i + 1$ of the inductive construction we divide the entire set of processors into $p^{\frac{i+2}{i+1}}$ sub-sets, $S_1, \ldots, S_{p^{\frac{i+2}{i+1}}}$, each of size $p^{\frac{i+2}{i+1}}$. Each subset $S_j$ separately performs:

1. Recursive application cooperative-sharing of the input values within itself. By the inductive hypothesis this takes $(p^{\frac{i+2}{i+1}})^{\frac{i+2}{i+1}} (c \log n \log^* p)^{i} = p (c \log n \log^* p)^{i}$ work.

2. $\frac{i+2}{i+1}$-scatter to all other processors. By construction this takes $< pc' (c \log p \log^* p)^{i+1}$ work.

3. $\frac{i+2}{i+1}$-gather from all other processors. By construction this takes $< pc' (c \log p \log^* p)^{i+1}$ work.

Thus, the total complexity is

$$< (2c' + 1)p^{\frac{i+2}{i+1}} (c \log p \log^* p)^{i+1} \leq p^{\frac{i+2}{i+1}} (c \log p \log^* p)^{i+1},$$

for $c$ such that $2c' + 1 \leq 1$.

We have obtained:

**Theorem 4.1.** There exists a constant $c$, such that for any $i < \log p$, cooperative-sharing is completed in

$$p^{\frac{i+2}{i+1}} (c \log p \log^* p)^{i}$$

(For $i \geq \log p$, there is no way to divide the processors into sets of size $p^{\frac{i+2}{i+1}}$.) The optimal value for $i$ is attained at $i + 1 \approx \sqrt[p]{\log p \log^* p}$. Plugging in this value in Theorem 4.1, and simplifying (sacrificing optimality for simplicity), gives Theorem 1.2.

**References**


Appendix.

A The Formal Model.

We assume a shared memory asynchronous system with $n$ processors, $P_1, \ldots, P_n$, and atomic read/write registers. The schedule of the system is described (posteriori) as a function $S: \mathcal{N} \to \{1, \ldots, n\}$, where $S(\pi) = j$ means that processor $P_j$ performed the $\pi$-th step in the system. A processor is never scheduled once it has completed its part in the protocol. The schedule function is assumed to be determined by an adversary scheduler. We assume a weak dynamic adversary scheduler model. The weak adversary scheduler comes in several incarnations, with some variation between them. In this work we follow the [11] definition, which they call the value oblivious adversary model. The model assumes a dynamic adversary scheduler, which at each step decides on the next processor to operate. For this decision, the adversary is provided with the entire history of the dynamics of the system. However, it is assumed that the adversary has no knowledge of the actual value stored in a memory location as long as no processor performs any operation based on this value. Thus, the value is hidden from the adversary as long as it is only written or read. It is immediately available to the adversary if any processor makes any decision based on the value, e.g., a conditional branch or a decision of which memory location to access. Furthermore, the model allows the adversary to know the value of a register one step before any processor is about to act upon the value. The rational behind this adversary model is that the asynchronous behavior of a system (e.g., system failures, page faults, etc.) is affected by what operations the processors perform, or the locations from which processors read and write, but not by the actual values being manipulated. Once a value is being operated upon, it may be used in ways which can affect the dynamic behavior of the system, and thus becomes available to the scheduler. The algorithm and the initial input values are assumed to be known to the adversary. A full description of the model can be found in [11].