COMMUNICATION BY NON-WRITING IN SYNCHRONOUS PARALLEL MACHINES

Or: The Joys of Not Writing

James E. Burns, Kurt W. Keutzer and Paul W. Purdom

Indiana University, Computer Science Department, Bloomington, Indiana

TECHNICAL REPORT NO. 140

COMMUNICATION BY NON-WRITING

IN SYNCHRONOUS PARALLEL MACHINES

Or: The Joys of Not Writing

by J. E. Burns, K. W. Keutzer and P. W. Purdom

June, 1983

Research reported herein was supported in part by the National Science Foundation under grants numbered MCS 79-06110 and MCS 80-04337.

COMMUNICATION BY NON-WRITING IN SYNCHRONOUS PARALLEL MACHINES

Or: The Joys of Not Writing

James E. Burns

Kurt W. Keutzer

Paul W. Purdom

Indiana University, Computer Science Department, Bloomington, Indiana

Abstract

Cook and Dwork recently described an algorithm for computing the Boolean OR of N bits on a PRAM in less than the "obvious" lower bound of $\log_2 N$ steps [CD]. Their algorithm makes use of an interesting technique of communicating information by *not* writing as well as by writing. We show that this technique can be used to speed up communication in many circumstances. In particular, arbitrary N-ary functions (over finite domains) can be computed in about $0.720 \log_2 N + 2.336 + O(e^{-N})$ steps on a PRAM (which is just two more step than Cook and Dwork use for computing Boolean OR).

The lower bound given by Cook and Dwork for computing OR-like functions on a PRAM is about $0.442 \log_2 N + 0.121 + O(e^{-N})$. We provide a simpler proof and an improved bound of about $0.526 \log_2 N + 0.526 + O(e^{-N})$. We also give a tight upper and lower bound of two steps for computing functions on the much more powerful WRAM model.

Another problem of interest in the PRAC model of computation is broadcasting a single data value from one cell to N. We provide a provably optimal algorithm for broadcasting a binary value in approximately $0.526 \log_2 N + 0.180 + O(e^{-N})$ steps.

1 Introduction

Two fundamental problems in parallel systems of computation are the centralization of information which is disbursed among the components and the broadcast of information from a single point to many. Cook and Dwork [CD] showed that the analysis of these problems is more subtle than might first be expected. In particular, the "obvious" lower bound of $\log_2 N$ steps to compute the Boolean OR of N bits in distinct memory cells in a PRAM can be beaten. Their technique takes advantage of the fact that, on a synchronous machine, information can be transferred by *not* writing (which we call *non-writing*) as well as by writing. The main thrust of this paper is to examine the implications of this technique for other problems and models.

The next section of the paper defines the shared memory machine model of computation, which is equivalent to the model used by Cook and Dwork, and the two problems

which we will investigate: the collection and broadcasting of information. Section three presents upper and lower bounds for the collection of information for WRAM and PRAM models, and optimal upper and lower bounds for the broadcast problem for the PRAC model. The final section summarizes our work and suggests future directions.

2 The SMM Model

We define a *shared memory machine* or *SMM* as a collection of processors which operate synchronously in parallel and communicate only through a common global random memory. In one *step*, each process reads at most one memory cell, does some local computing, and then writes into at most one memory cell. The information known to a processor is included in its *state*, and the information contained in a cell is given by its *value*. We will denote processes by p and q, perhaps subscripted, and memory cells by m_i for positive integers i.

We will discuss three different models of SMM's which differ in the kinds of concurrent access allowed to memory cells. A *WRAM* is a SMM which allows concurrent reads and concurrent writes. We follow Shiloach and Vishkin [SV] in requiring that if concurrent writes to a single cell occur, all writing processes must write the same value; however, we note that our lower bound also applies to less restrictive concurrent write models. A *PRAM* [FW] is a SMM which allows concurrent reads, but only exclusive writes. A *PRAC* [LPV] is a SMM which allows only exclusive reads and writes of cells in one step.

We will use the variable t to refer to the steps of a computation. Initially, $t = 0$, every process is in a distinguished initial state, and all but a finite number of memory cells have the special value λ. The non-λ values of memory are called the *input* of the program which the system executes. The transition from t to $t+1$, (referred to as $t \rightarrow t+1$) is broken into a read part and a write part. Note that the states of processors change only in the read part, and the values of memory cells change only in the write part. Thus when we say that "at step $t+1$ process p writes into cell m_i", we mean that process p writes into cell m_i in the write part of transition $t \rightarrow t+1$.

We will analyze the complexity of two problems called the *collection problem* and the *broadcast problem*. The collection problem involves computing a value which depends on the values in the input, while the broadcast problem involves the copying of information from one cell to many.

The collection problem essentially involves the computation of a function of the input. The difficulty of collecting enough information to compute the function may depend strongly on the function being computed; some functions are trivial to compute. To obtain non-trivial lower bounds, we define a (slightly) restricted class called *sensitive functions* to which our lower bound for the PRAM model applies and a more restricted class called *dependent functions* to which our lower bound for the WRAM model applies.

Let f be an N-ary function. A SMM is said to *compute f in T steps* if for all $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ in the domain of f, the computation beginning at $t = 0$ with $m_i = x_i$ for $1 \leq i \leq N$ and $m_i = \lambda$ for $i > N$, the computation terminates with $m_1 = f(\mathbf{x})$ at $t = T$.

For any $\mathbf{x}$ and $\mathbf{y}$, define $S(\mathbf{x}, \mathbf{y}) = \{\mathbf{z} : \text{for all } i,\ 1 \leq i \leq N,\ z_i \in \{x_i, y_i\}\}$. We say that f is *normal* if for every $\mathbf{x}$ and $\mathbf{y}$ in the domain of f, $S(\mathbf{x}, \mathbf{y})$ is a subset of the domain of f. A normal function has the property that the values of the elements of the vectors in its domain do not depend on one another. From now on we will assume that all functions that we will be dealing with are normal.

We say that f is *sensitive at $\mathbf{x}$ relative to $\mathbf{y}$* if for all i, $1 \leq i \leq N$, $f(\mathbf{x}) \neq f(x_1, x_2, \ldots, x_{i-1}, y_i, x_{i+1}, \ldots, x_N)$. For example, the Boolean OR function for N bits is sensitive at $(0, 0, \ldots, 0)$ relative to $(1, 1, \ldots, 1)$. Also, f is *dependent for $\mathbf{x}$ and $\mathbf{y}$* if for every $\mathbf{z} \in S(\mathbf{x}, \mathbf{y})$ and for every $1 \leq i \leq N$, $f(z_1, z_2, \ldots, z_{i-1}, x_i, z_{i+1}, \ldots, z_N) \neq f(z_1, z_2, \ldots, z_{i-1}, y_i, z_{i+1}, \ldots, z_N)$. We also say a that a function is *sensitive* or *dependent* if it is sensitive or dependent, respectively, for some $\mathbf{x}$ and $\mathbf{y}$. Note that the parity function is dependent, but the Boolean OR function is not.

A SMM is said to solve the *broadcast problem for N cells in T steps* if, beginning at $t = 0$ with $m_1 = b$ (for some $b \neq \lambda$) and $m_i = \lambda$ for all $i > 1$, every cell $m_j = b$ for $1 \leq j \leq N$ for all times $t \geq T$. The complexity of this problem (for some models) depends on the range of values which b may take on.

3 Upper and Lower Bounds

It is obvious that the WRAM is the most powerful of the models defined in the previous section, since it subsumes the others. In fact, in the WRAM model, only two steps are necessary to compute any function with finite domain, as shown by the next theorem. The (reasonably obvious) algorithm used in the proof will also be useful in the

proof of Theorem 3. We also note that this algorithm is tight for general functions since any dependent function with $N > 2$ cannot be computed in less than two steps.

Theorem 1: Any N-ary function with finite domain can be computed in two steps on a WRAM.

Proof: Let b be an integer large enough so that we can code each element of each vector in the domain of f with the values $0, 1, \ldots, b$. Let $\mathbf{x}$ be any input, and let V be its value coded as a base b number.

In the first step, for each i and j, $1 \leq i \leq N$, $0 \leq j \leq b^{N-1}$, process $p_{i,j}$ reads memory cell m_i. If the value read is *not* equal to the i^{th} digit of j (considered as a base b number), then $p_{i,j}$ writes a 1 into cell m_{N+j+1}; otherwise $p_{i,j}$ does not write. Thus, the only cell in $\{m_{N+1}, m_{N+2}, \ldots, m_{N+b^N}\}$ which has value λ is m_{N+1+V}.

In the second step, process $p_{1,j}$ reads memory cell m_{N+1+j} for $0 \leq j \leq b^N - 1$. The unique process $(p_{i,N+1+V})$ which reads λ then determines $\mathbf{x}$ and writes the value of $f(\mathbf{x})$ into m_1. □

It may be objected that the algorithm in the above proof uses an exponential number of processes. However, the following result shows that an exponential number of processes are required, in general, if we wish to compute the answer in two steps.

Theorem 2: Let f be any N-ary function, $N > 2$, such that for any $\mathbf{x}$ and $\mathbf{y}$ in the domain of f with $\mathbf{x} \neq \mathbf{y}$, $f(\mathbf{x}) \neq f(\mathbf{y})$. Any WRAM which computes f in two steps must use at least 2^{N-2} processes.

Proof: We may assume without loss of generality that f is a function of binary inputs. By assumption, every one of the 2^N possible inputs must leave a distinct result in m_1 at the end of two steps. On the first step, each process learns the value of at most one bit of the input. Any process which reads a cell written by another process in the first step can learn about at most one other bit of the input. But, since $N > 2$, such a process cannot know the entire input and hence cannot invariably write the correct value into cell m_1 on the write part of step 2.

Any process which reads a cell on the second step which was not written on the first step can be in one of at most four states at the second step and must behave in one of

four ways. But since there are less than 2^{N-2} processes, there are less than 2^N possible values written into m_1 on the second step. This is insufficient because f has 2^N possible results. □

Cook and Dwork [CD] give an algorithm for computing the Boolean OR of N bits in $c_1 \log_2 N + c_2 + O(e^{-N})$ steps, where $c_1 = 1/\log_2[(3+\sqrt{5})/2] \approx 0.720$ and $c_2 = \log_2[(5-\sqrt{5})/2]/\log_2[(3+\sqrt{5})/2] \approx 0.336$. We use their algorithm as a subroutine and the coding technique of Theorem 1 to get the following result.

Theorem 3: Let f be any dependent N-ary function with finite domain, $N > 1$. Then f can be computed in $2 + c_1 \log_2 N + c_2 + O(e^{-N})$ steps, where c_1 and c_2 are given above.

Proof: Let $\mathbf{x}$, b and V be as in the proof of Theorem 1. The algorithm consists of a preliminary step similar to the algorithm in Theorem 1, b^N copies of Cook and Dwork's algorithm run in parallel, and a final clean-up step to write the answer into m_1.

In the preliminary step, process $p_{i,j}$ reads memory cell m_i for $1 \le i \le N$ and $0 \le j \le b^N - 1$. If the value read is not equal to the i^{th} digit of j (considered as a base b number), then $p_{i,j}$ writes a 1 into $m_{N+(i-1)b^N+1+j}$. This is similar to the algorithm in Theorem 1 except that it clearly avoids multiple writes to the same cell.

The main part of the algorithm consists of b^N independent computations. For each j, $0 \le j \le b^N - 1$, we use Cook and Dwork's algorithm to compute the Boolean OR of memory cells $m_{k_1}, m_{k_2}, \ldots, m_{k_N}$ into cell m_{k_1}, where $k_i = N + (i-1)b^N + j + 1$ for $1 \le i \le N$. (We take λ as equivalent to 0.) At the end of this part of the algorithm, there is exactly one cell in $m_{N+1}, \ldots, m_{N+b^N}$ which has a value of λ.

In the final step of the algorithm, process $p_{1,j}$ reads memory cell m_{N+1+j} for $0 \le j \le b^N - 1$. The single process $(p_{1,N+1+V})$ which reads value λ then determines $\mathbf{x}$ and writes $f(\mathbf{x})$ into m_1. □

Once again, we have been very extravagant with processors. An interesting area for further research is to characterize functions by the number of processors that they require in order to be computed quickly in this model. We believe that the semigroups may be a fruitful area for further work; Chandra, Fortune and Lipton have investigated the complexity of computing semigroup products on circuits with unbounded fan-in [CFL].

Cook and Dwork [CD] proved that no sensitive N-ary function can be computed in less than approximately $0.442 \log_2 N + 0.121 + O(e^{-N})$ by a PRAM. The following result improves this bound and gives a simpler proof. Note that the gap (now about $0.194 \log_2 N$) between the upper and lower bounds is still not closed. We feel that it can be narrowed further.

Theorem 4: Let f be any sensitive N-ary function. The a PRAM requires at least $c_3 \log_2 N + c_4 + O(e^{-N})$ steps, where $c_3 = 1/\log_2(2+\sqrt{3}) \approx 0.526$ and $c_4 = c_3$.

Before attacking the proof we will need some definitions, (essentially due to Cook and Dwork) and two lemmas. In the following, we refer to a function f which is sensitive to $\mathbf{x}$ relative to $\mathbf{y}$ and to a computation with input $\mathbf{x}$. We also use the notation $\mathbf{x}(i) = (x_1, x_2, \ldots, x_{i-1}, y_i, x_{i+1}, \ldots, x_N)$.

Definition: An index i *affects* process p or a memory cell m_k at time t if the state of p or the value in cell m_k, respectively, at the end of step t is different for inputs $\mathbf{x}$ and $\mathbf{x}(i)$.

Let $Affects_P(p,t)$ and $Affects_M(m_k,t)$ be the sets of indices which affect p and m_k, respectively, at time t.

Definition: Index i *causes* process p to write into cell m_k at step t iff p writes into m_k in the transition $t-1 \rightarrow t$ with input $\mathbf{x}(i)$ and p does not write into m_k with input $\mathbf{x}$.

If an index causes a process to write a cell, we can deduce something about the original value of the corresponding memory cell when a write does *not* occur. We let the set of indices which cause p to write m_k at step t be denoted by $Causes(p, m_k, t)$. The union of $Causes(p, m_k, t)$ for all p and a given m_k and t is given by $Causes(m_k, t)$. Observe that we may be able to deduce information about indices other than those in the causing set. For example, some process may write only under the condition that *two* of the elements of $\mathbf{x}$ are changed, and these are not necessarily part of the causing set. Nevertheless, the definition as stated is sufficient to prove our results.

Definition: A process p *dominates* a process q with respect to cell m_k at step t iff $Causes(q, m_k, t) \subseteq Affects_P(p, t)$.

That is, one process dominates another iff it is affected by all the indices which cause the other process to write. We now give two lemmas which will allow us to prove the theorem.

Lemma 1: For any PRAM which satisfies Theorem 4, any cell m_k, any time $t > 0$, and any processes p and q, either p dominates q or q dominates p with respect to m_k.

Proof: If $p = q$, it is clear from the definitions that $Causes(p, m_k, t) \subseteq Affect_P(p, t)$, (otherwise a process would have distinct behavior from the same state). Suppose $p \neq q$.

Let i_p be an index in $Causes(p, m_k, t) - Affects_P(q, t)$ and let i_q be an index in $Causes(q, m_k, t) - Affects_P(p, t)$. But then both p and q will write m_k from $\mathbf{x}(i_p)(i_q)$, which is not allowed in an exclusive write model. □

Lemma 2: For all processors p, memory cells m_k and times t, $|Affects_P(p, t)| \leq P_t$ and $|Affects_P(m_k, t)| \leq C_t$, where P_t and C_t are given by the following recurrence.

(1) $P_0 = 0$

(2) $C_0 = 1$

(3) $P_{t+1} = P_t + C_t$

(4) $C_{t+1} = C_t + 2P_{t+1} - 1$

Proof: By induction on t. The base case for $t = 0$ is clear.

For the induction step, a processor p can be affected by an index i at step $t+1$ only if it was affected at step t or if it read a cell m_k which was affected by i at step t. Therefore, by the induction hypothesis and (3), $|Affects_P(p, t+1)| \leq P_t + C_t = P_{t+1}$.

If some process p writes cell m_k at step $t+1$, then the only indices which affect m_k at $t+1$ are those which affect p at $t+1$, so $|Affects_M(m_k, t+1)| \leq P_{t+1} \leq C_t + 2P_{t+1} - 1$ since P_t is non-decreasing and $P_1 = 1$. We therefore only need to consider the case where no process writes into m_k in step $t+1$.

Suppose that no process writes into m_k in step $t+1$. Then the only indices that can affect m_k at step $t+1$ are those which affected m_k at step t and those which are in $Causes(m_k, t+1)$. By the induction hypothesis therefore, $|Affects_M(m_k, t+1)| \leq C_t + |Causes(m_k, t+1)|$. We will demonstrate that $|Causes(m_k, t+1)| \leq 2P_{t+1} - 1$, which gives the lemma.

Suppose that $|Causes(m_k, t+1)| \geq 2P_{t+1}$. Let $r = |Causes(m_k, t+1)|$ and let $Causes(m_k, t+1) = \{u_1, u_2, \ldots, u_r\}$. Note that for each u_i there is a unique process q_i such that q_i is caused to write cell m_k at step t by index u_i. Let G be a directed graph

with r nodes labels with $u_1, u_2, \ldots, u_r$, and let and edge (u_i, u_j) be in G iff process q_j dominates q_i. Note that if (u_i, u_j) is an edge in G, then u_i must affect q_j. By Lemma 1, G must have at least $r + r(r-1)/2 = r(r+1)/2$ edges. But then some node, u_i, must have at least $\lceil (r+1)/2 \rceil > P_{t+1}$ incoming edges. By our construction, this implies that $|Affects_P(q_i, m_k, t+1)| > P_{t+1}$, contrary to assumption. □

Proof of Theorem 4: Solving the recurrence equations in Lemma 2 gives

$$C_t = \tfrac{1}{2}(2+\sqrt{3})^t + \tfrac{1}{2}(2-\sqrt{3})^t$$

Since f is sensitive at $\mathbf{x}$ we must have $\{1, 2, \ldots, N\} \subseteq Affects_M(m_1, T)$. Therefore, any PRAM algorithm computing f in T steps must have $N \leq |Affects_M(1, T)| \leq C_T = \frac{1}{2}(2+\sqrt{3})^T + \frac{1}{2}(2-\sqrt{3})^T$. Solving this equation for T in terms of N gives us the formula required by the theorem. □

We now turn to a somewhat symmetric problem, the broadcasting of a single datum to N memory cells. This problem is trivial if we allow concurrent reads (it can be done in a single step); we will therefore examine only PRACs.

There is an straightforward algorithm (without using non-writing) for broadcasting an item of information to N cells in about $0.720 \log_2 N + 0.336 + O(e^{-N})$ steps (it happens to have the same recurrence as the Cook and Dwork Boolean OR algorithm). However, if the value is binary and non-writing is used, we can do much better.

Theorem 5: The problem of broadcasting a binary value to N cells can be solved in $c_5 \log_2 N + c_6 + O(e^{-N})$ steps on an PRAC, where $c_5 = 1/\log_2(2+\sqrt{3}) \approx 0.526$ and $c_6 = 1 - \log_2[(3+\sqrt{3})/6]/\log_2(2+\sqrt{3}) \approx 0.180$.

Proof sketch: The basic idea of the algorithm is that on the write phase, each process which knows the broadcast value writes into one of two previously unaffected cells. If the value is 0, then 0 is written into the one cell, otherwise a 1 is written into the other cell. On the next read phase, a new process is activated for each affected cell (whether written or not written), and the new processes can deduce the broadcast value. This gives a significant speed up over the obvious algorithm which only activates processes by having them read cells which have actually been written into.

In order to achieve optimality, we must use one more trick. On the first step, we use $N-3$ processors (which do not yet know the broadcast value) to write into all the cells that are not involved in the first step of the algorithm. These processors write 0 into the odd indexed cells and 1 into the even indexed cells, excluding m_1 and the two cells which are affected by the process reading m_1 in the first step. This saves at most one step, but is necessary if we are to match the lower bound. □

The upper bound is tight, as shown by the next theorem.

Theorem 6: Any algorithm solving the problem of broadcasting a binary value to N cells must take at least $c_5 \log_2 N + c_6 + O(e^{-N})$ steps, where c_5 and c_6 are the same as in Theorem 5.

Proof sketch: The proof technique is similar to that used in Theorem 4. We give a recurrence which limits the number of cells which can be affected by the first cell after t steps. Since all of the first N cells must be affected at the conclusion of the algorithm, solving this recurrence gives us the result. Some care must be taken to show that this bound is an exact match for the upper bound. □

4 Summary

We have shown that the non-writing technique can be used to improve over the obvious algorithms for various problems and models of parallel computation. Whenever optimal algorithms are sought for synchronous computation, this technique should be carefully considered.

The technique of non-writing can also be applied to limited connection machines such as those discussed by Siegel [S] and Galil and Paul [GP]. In this model each processor has some local memory and is connected to other processors by an interconnection network that is either fixed for all N or limited to some sparse number of connections. By using non-writing on this model, the problem of broadcasting data to N processors can be solved in $\log_3 N$ steps. This bound is non-trivial and can be shown to be optimal for limited connection machines in which each processor can only write to one other processor at a step. A result similar to that of Theorem 4 can be derived for limited connection

machines but a non-writing technique has not been able to supply an upper bound that beats the obvious $\log_2 N$.

One interesting area for future investigation is the characterization of functions by the amount of processing power that they require to be computed quickly. For example, the Boolean OR function can be computed quickly on a PRAC with only a linear number of processors, while an arbitrary function apparently requires an exponential number. What can be said about functions more complex than OR?

References

[CFL] Chandra, A.K., S. Fortune, and R. Lipton. Unbounded fan-in circuits and associative functions. Proc. of the 15^{th} Annual ACM Symposium on the Theory of Computing, Apr. 1983, p. 52–60.

[CD] Cook, S., and C. Dwork. Bounds on the time for parallel RAM's to compute simple functions. Proc. of the 14^{th} Annual ACM Symposium on the Theory of Computing, May 1982, p. 231–233.

[FW] Fortune, S., and J. Wylie. Parallelism in random access machines. Proc. of the 10^{th} Annual ACM Symposium on the Theory of Computing, May 1978, p. 114–118.

[LPV] Lev, G., N. Pippenger, and L. Valiant. A fast parallel algorithm routing in permutation networks. *IEEE Trans. on Comput. C-30*, 2 (1981) 93–100.

[GP] Galil, Z., and W.J. Paul. An efficient general-purpose parallel computer. *JACM 30*, 2 (1983) 360–387.

[SV] Shiloach, Y., and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. Dept. of Comp. Sci., Technion Isreal, TR173, Mar. 1980.

[S] H.J. Siegel. A model of SIMD machines and comparison of various interconnection networks. *IEEE Trans. on Comput. C-28*, 12 (1979) 907–917.