MAX-PLANCK-INSTITUT FÜR INFORMATIK

Sensitive Functions and Approximate Problems

S. Chaudhuri

MPI–I–93–145 Oktober 1993

INFORMATIK

Im Stadtwald
66123 Saarbrücken
Germany

Sensitive Functions and Approximate Problems

S. Chaudhuri

MPI-I-93-145 Oktober 1993

Sensitive Functions and Approximate Problems

Shiva Chaudhuri
Max-Planck-Institut für Informatik
Im Stadtwald
6600 Saarbrücken
Germany
E-mail: shiva@mpi-sb.mpg.de

Abstract

We investigate properties of functions that are good measures of the CRCW PRAM complexity of computing them. While the block sensitivity *is known to be a good measure of the CREW PRAM complexity, no such measure is known for CRCW PRAMs. We show that the complexity of computing a function is related to its* everywhere sensitivity, *introduced by Vishkin and Wigderson. Specifically we show that the time required to compute a function $f : D^n \longrightarrow R$ of everywhere sensitivity $es(f)$ with $P \geq n$ processors and unbounded memory is $\Omega(\log[\log es(f)/(\log 4P|D| - \log es(f))])$. This improves previous results of Azar, and Vishkin and Wigderson. We use this lower bound to derive new lower bounds for some* approximate problems. *These problems can often be solved faster than their exact counterparts and for many applications, it is sufficient to solve the approximate problem. We show that* approximate selection *requires time $\Omega(\log[\log n/\log k])$ with kn processors and* approximate counting *with accuracy $\lambda \geq 2$ requires time $\Omega(\log[\log n/(\log k + \log \lambda)])$ with kn processors. In particular, for constant accuracy, no lower bounds were known for these problems.*

1 Introduction

The computation of Boolean functions by circuits leads naturally to their study in all models of parallel computation. Much work has been done on investigating properties of Boolean functions which are measures of the difficulty of computing the function. One such measure is the *sensitivity* of a function. Let $f : \{0,1\}^n \rightarrow \{0,1\}$ be a Boolean function, let $i \in \{0,1\}^n$ and let $i^{(r)} \in \{0,1\}^n$ denote the bit vector that differs from i exactly on the rth co-ordinate. Then the sensitivity of f on i, written $s_f(i)$ is the number of distinct co-ordinates r, such that $f(i) \neq f(i^{(r)})$. The sensitivity of f, written s_f, is the maximum, over all inputs i, of $s_f(i)$. The sensitivity of Boolean functions has been extensively studied [15, 16, 6, 14]. Cook, Dwork and Reishuk [6] show that the complexity of computing a function f on a CREW PRAM is related to the sensitivity of f. They prove a lower bound of $\Omega(\log s_f)$ on the time required to compute f. Nisan considered a generalization of sensitivity, called the *block sensitivity* [14]. He showed that the time required to compute a function f on a CREW PRAM is $\Theta(\log bs_f)$, where bs_f is the block sensitivity of f. Thus, the complexity of computing Boolean functions on CREW PRAMs is well characterized.

The AND function has sensitivity n and therefore takes $\Theta(\log n)$ time on a CREW PRAM. However, AND can be computed in constant time on a CRCW PRAM. Thus, sensitivity and block sensitivity are not appropriate measures of CRCW PRAM complexity. The reason is that both measures can critically depend on the value of the function on a single input. A CRCW PRAM can use its concurrent writing property to check, in a single step, if its input is special. Thus any measure whose value depends on a small number of inputs is doomed to failure.

A measure that avoids dependence on a small set

on inputs is *everywhere sensitivity*, defined as follows. Let D and R be finite sets and let $f : D^n \rightarrow R$ be a function. An input $i \in D^n$ is $q-sensitive$ if for every subset $J \subseteq \{1,\ldots,n\}$, $|J| = q-1$, there exists an input l, which agrees with i on the co-ordinates in J and for which $f(i) \neq f(l)$. The everywhere sensitivity of i is the largest integer q such that i is $q-sensitive$. The everywhere sensitivity of f is the *minimum*, over all inputs i, of the everywhere sensitivity of i. An alternative way of thinking of the everywhere sensitivity of a function is the maximum number of co-ordinates whose values can be safely revealed without revealing the value of the function. Vishkin and Wigderson showed that a CRCW PRAM with m memory cells requires time $\Omega(\sqrt{q/m})$ to compute a function of everywhere sensitivity q [17]. For a special class of functions, Azar improved this bound to $\Omega(q/m)$ [1]. However, this does not yield nontrivial bounds for $m = \Omega(n)$.

We investigate the role of everywhere sensitivity in determining the CRCW PRAM complexity of a function. Our main result is that computing a function $f : D \longrightarrow R$ of everywhere sensitivity $es(f)$ requires time $\Omega(\log[\log es(f)/(\log 4P|D| - \log es(f))])$ on a CRCW PRAM with $P \geq n$ processors and unbounded memory. The lower bound holds for nonuniform algorithms as well. For computing, with n processors, a Boolean function of everywhere sensitivity n, for instance, PARITY, this gives a lower bound of $\Omega(\log\log n)$. This is weaker than the bound of $\Omega(\log n/\log\log n)$ obtained by Beame and Håstad [3]. However, surprisingly, for n processors and everywhere sensitivity $\Omega(n)$, the bound is tight for nonuniform algorithms. In Section 5.1 we give an example of a function with everywhere sensitivity $\Omega(n)$ which can be computed in $\mathcal{O}(\log\log n)$ time with n processors.

As applications of the above bound we derive new lower bounds for other problems. These problems have the common feature that they are all *approximate* versions of problems. For some applications, it is enough to solve the approximate version, which can often be solved faster than the exact version [9, 10, 5, 13]. For each approximate problem, there is an *accuracy parameter* $\lambda \geq 1/(n+1)$. In approximate selection, the task is to find, from n elements, an element whose rank differs from a specified rank by at most λn. In approximate counting, given a bit vector, the goal is to compute an integer that lies between $s/(\lambda+1)$ and $s(\lambda+1)$, where s is the number of 1's in the input vector. We prove the following bounds: approximate selection with accuracy $\lambda \leq 1/4$ with Cn processors requires $\Omega(\log[\log n/\log C])$ time. Approximate counting with accuracy $\lambda \geq 2$ using Cn processors requires $\Omega(\log[\log n/(\log\lambda + \log C)])$ time. These bounds are easily seen to imply lower bounds for other approximate problems such as *interval allocation* and *approximate prefix summation* [7, 10]. In particular, the bound for approximate counting directly implies the bound for *approximate compaction* proved in [4].

Padded sorting is the problem of placing n elements in sorted order in an output array of size at most $(\lambda+1)n$. The unused locations should contain a special *null* value. We mention here that our bounds for everywhere sensitive functions can also be used to derive a lower bound of $\Omega(\log(\log n/(\log\lambda+\log C)\log\lambda))$ for padded sorting with accuracy $\lambda \geq 2$ using Cn processors.

When $\lambda = 1/(n+1)$ each of the above three problems reduces to its exact version, which is known to require $\Omega(\log n/\log\log n)$ time. It has been shown that if one can solve any of the approximate problems with an accuracy of λ using p processors in time t, then one can, with the same resources, solve a MAJORITY problem on $1/\lambda$ elements [11]. The lower bound of Beame and Hastad then implies an $\Omega(\log(1/\lambda)/\log\log n)$ time lower bound for solving any of these problems with a polynomial number of processors. While this is good for small values of λ, it does not give a nontrivial bound when $\lambda = \Omega(1/(\log n)^c)$, for constant c. Our bounds improve the above for this range of λ, where, in fact, fast algorithms are known for each of the problems. In particular, for $\lambda = 1/4$, with n processors, algorithms for approximate selection and approximate counting are known that run in time $\mathcal{O}((\log\log n)^4)$ and $\mathcal{O}((\log\log n)^3)$ respectively [5, 7]. With n^2 processors padded sorting can be solved in $\mathcal{O}((\log\log n)^4)$ time [10].

The methods used to prove the lower bounds are of independent interest. In Section 3 we prove some lemmas applicable to *any* PRAM algorithm. Roughly speaking, these lemmas state that it is possible to bound the number of possible states of an algorithm by carefully setting a small number of inputs. This makes it possible to prove a lower bound for any problem, merely by showing that if a small number of inputs is set, a large number of output possibilities still remain. The method of bounding the number of states is general enough to have applications to other computational models.

2 Preliminaries

2.1 The Model

We prove the lower bound on a strong model of the PRIORITY CRCW PRAM (see [12]). This model is sometimes referred to as the "Ideal" or "Full-Information" PRAM [2]. In this model, each processor is assumed to keep track of the entire history of its own computation. Each processor has an *initial state*, and its state at step i is defined by its initial state and it history through step $i-1$. We consider deterministic algorithms, so the action of a processor is completely determined by its state. We make no other assumptions about the algorithm, in particular, nonuniform algorithms are allowed.

We assume an infinite number of memory cells with infinite wordsize. Since there is no restriction on the wordsize, whenever a processor writes, it may as well write the entire history of its computation; hence the name Full-Information PRAM. Lower bounds on this model depend crucially on limiting the amount of information that processors can communicate to each other through the shared memory. Lower bounds proved on this model carry over to more realistic models and give insight into the intrinsic difficulty of solving problems in parallel.

2.2 Partial Inputs and the Computation Graph

In the following, A will be an algorithm computing a function $f : D^n \longrightarrow R^s$. For inputs of size n, let A use $P(n)$ processors and take $k(n)$ steps. We will use P and k for $P(n)$ and $k(n)$ respectively, from now on.

A *partial input* is an element of $(D \cup \{*\})^n$. For a partial input b, we denote by $X(b)$ the set of inputs consistent with b. That is, $X(b) = \{x \in D^n : \textit{for } i = 1,\ldots,n,\ b_i \neq * \rightarrow b_i = x_i\}$. For partial inputs a and b, we say a is a refinement of b, and write $a \leq b$, if $X(a) \subseteq X(b)$.

For a given partial input b, consider a processor p at time t. The set of inputs consistent with b defines a set of states that p may be in, on inputs consistent with b. This set of states, in turn, defines the possible actions of p at time t, in particular, it defines the set of memory locations that p may read from, or write to. This gives us a way to identify (and consequently, limit) the amount of information that p may read from, or write to, the shared memory. We formalize this by modelling the computation of A on a graph. Let b be a partial input of size n. The *computation graph* of A on b, $G(b)$, is defined as follows.

$$V(G(b)) = \{(c,i) : c \textit{ is a memory cell and } 0 \leq i \leq k\}.$$

That is, we have $(k+1)$ levels; in each level we have one vertex for each cell in the memory. The set of vertices in level i will be called V_i. The directed edges go from vertices at one level to the vertices of the next level. Every edge is labelled by a processor. $E(G(b))$ contains the edge $((c,i),(d,i+1))$ labelled p if on some input in $X(b)$, processor p reads cell c and writes to cell d in step $(i+1)$. Initially, variable i of the input is assumed to be in cell i; finally, output value i is assumed to be in cell i. We refer to vertex $(i,0)$ as α_i for $1 \leq i \leq n$ (the *input* vertices), and vertex (i,k) as β_i for $1 \leq i \leq s$ (the *output* vertices).

Let $a \in D^n$. We shall associate with each vertex of $G(a)$ a content. The content associated with (c,i) is the content of the cell c after step i (that is, just before the write of step $(i+1)$ changes it) in the computation of A on the input a. We call this content $\textit{content}(a,(c,i))$. Similarly, for a processor p, $\textit{state}(a,(p,i))$ is the state of processor p just before the write of step $(i+1)$ in the computation of A on input a. For a partial input b, let

$$\begin{aligned} \textit{contents}(b,(c,i)) &= \{\textit{content}(x,(c,i)) : x \in X(b)\}; \\ \textit{states}(b,(p,j)) &= \{\textit{state}(x,(p,i)) : x \in X(b)\}. \end{aligned}$$

We say that (c,i) ((p,i)) is a *fixed* vertex (processor) if $|\textit{contents}(b,(c,i))| = 1$ ($|\textit{states}(b,(p,i))| = 1$); otherwise we say (c,i) is a *free* vertex (processor). Note that the above definitions depend on the algorithm A and the size of input n. These parameters will be clear from the context where they are used.

We model the computation of the algorithm A on the computation graph as follows. We say that a processor p reads from cell (c,i) and writes to cell $(d,i+1)$ when we mean that in the step $(i+1)$ of the computation of the algorithm A, p reads cell c and writes to cell d.

Let b be a partial input and consider a processor p. Let $a, a^{(j)} \in D^n$ be inputs such that $a, a^{(j)} \leq b$ and they differ only on the jth co-ordinate. We say that input variable x_j *affects* p on b at step i if there exist $a, a^{(j)}$ as above such that $\textit{state}(a,(p,i)) \neq \textit{state}(a^{(j)},(p,i))$. We write $\textit{affect}(b,(p,i))$ for the set of variables that affect p on b at the ith step. In an analogous way we define $\textit{affect}(b,(c,i))$ for a cell c.

We make a straightforward observation about variables that may affect a processor or cell. On a given partial input, b, $\textit{states}(b,(p,i))$ is the set of possible

states that p may be in at step i, on b. The state of a processor determines which cell it reads, thus this set defines a set of memory cells that p may read at step i, on b.

Observation 2.1 *Let b be a partial input and let $c_1, \ldots, c_r$ be the cells that processor p may read at step i, on b. Then the set of input variables that may affect p at step i,* $\text{affect}(b,(p,i)) \subseteq [\cup_{j=1}^{r}\text{affect}(b,(c_j,i))] \cup \text{affect}(b,(p,i-1))$. *Similarly let $p_1, \ldots, p_q$ be the processors that may write to cell c in step i, on b. Then the set of input variables that may affect c at step i,* $\text{affect}(b,(c,i)) \subseteq [\cup_{j=1}^{q}\text{affect}(b,(p_j,i))] \cup \text{affect}(b,(c,i-1))$.

Proof. Let x_l be a variable that is in $affect(b,(p,i))$ but not in $[\cup_{j=1}^{r} affect(b,(c_j,i))] \cup affect(b,(p,i-1))$. By definition, there exist inputs $a, a^{(l)} \leq b$ such that they differ on exactly the variable x_l, and $state(a,(p,i)) \neq state(a^{(l)},(p,i))$. On the other hand, since $x_l \notin [\cup_{j=1}^{r} affect(b,(c_j,i))] \cup affect(b,(p,i-1))$, $contents(a,(c_j,i)) \doteq contents(a^{(l)},(c_j,i))$ for each cell c_j and $state(a,(p,i-1)) = state(a^{(l)},(p,i-1))$. On both inputs, p is in the same state after $i-1$ steps, and will hence read the same cell, which has the same contents. Thus the history of p on both inputs is identical and hence $state(a,(p,i)) = state(a^{(l)},(p,i))$, a contradiction.

The statement about variables that may affect a cell may be proved similarly. ∎

3 Regularized Computation Graphs

Intuitively, if a cell is written to by a small number of processors, then it can only be affected be a small number of input variables, namely those that affect the processors that write to it. Similarly, it can only have a small number of contents, namely the ones that each processor may write. Thus computation graphs in which no cell is written to by many processors are of special interest, which motivates the following definition.

Definition: Let S be a sequence of positive integers $(d_0, d_1, d_2, \ldots)$. For a partial input b, we say $G(b)$ is *S-regularized upto level j* if every free vertex in $G(b)$ at level $i, 1 \leq i \leq j$ has indegree less than d_i. If $G(b)$ has k levels and is S-regularized upto level k, we simply say $G(b)$ is S-regularized. If $G(b)$ is S-regularized, then we call b an *S-regularizing* input.

The above definition implies that at level i, at most $d_i - 1$ processors may write to any free vertex. We would expect that this property ensures a bound (dependent on i) on the number of contents that a cell at level i may have. This is indeed true, as shown by the following lemma.

In a computation graph $G(b)$, define:

$$\begin{aligned} Y_i &= \max\{|contents(b,(c,i))| : c \text{ is a memory cell}\} \\ Z_i &= \max\{|states(b,(p,i))| : p \text{ is a processor}\} \\ M_i &= \max\{|affect(b,(c,i))| : c \text{ is a memory cell}\}; \\ N_i &= \max\{|affect(b,(p,i))| : p \text{ is a processor}\}. \end{aligned}$$

Lemma 3.1 *Let $S = (d_0, d_1, d_2, \ldots)$ be a sequence of positive integers. Let $\Delta_i = 2^{2^i}\Pi_{j=0}^{i} d_j^{2^{i-j}}$. Let G be a computation graph for an algorithm which takes inputs from D^n, where $|D| \leq d_0$. Suppose G is S-regularized upto level j. Then, for each i, $1 \leq i \leq j$,*

$$Y_i \leq d_i\Delta_{i-1}, \qquad Z_i \leq \Delta_i,$$
$$M_i \leq 2^{i-1}d_i\Delta_{i-1}, \qquad N_i \leq 2^i\Delta_i.$$

If, in addition, S satisfies $d_0 \geq 4$ and $d_{i+1} \geq d_i^4$ for $i \geq 0$, then $Y_i, Z_i, M_i, N_i \leq d_i^2$.

Proof. Consider a vertex (c,i) $(i > 0)$ in the graph $G(b)$. Let $d < d_i$ be the indegree of (c,i). Let $p_1, \ldots, p_d$ be the processors that label the d edges. Let the number of states in which processor p_j writes to (c,i) be S_j. The content of (c,i) is determined by the state of the processor that succeeds in writing to (c,i), or, if no processor writes to (c,i), by the content of $(c,i-1)$. Thus, we have

$$|contents(b,(c,i))| \leq \sum_{j=1}^{d} S_j + |contents(b,(c,i-1))|.$$

By definition, $S_j \leq Z_{i-1}, \forall j$ and $|contents(b,(c,i-1))| \leq Y_{i-1}$. Thus, for $i \geq 1$,

$$Y_i \leq (d_i - 1)Z_{i-1} + Y_{i-1}$$

The number of states of a processor after the ith read is at most the product of the number of states it had after the $i-1$th read and the number of contents of the cell it read at the ith read. Thus

$$Z_i \leq Z_{i-1}Y_i.$$

We have that $Y_0 = |D|$ and $Z_0 = |D|$, since, initially, each cell has a value in D and each processor, after the first read, can be in at most $|D|$ states. It can then be shown by induction on i that $Z_i \leq \Delta_i$ and $Y_i \leq d_i\Delta_{i-1}$.

Since the computation graph is regularized, a vertex at level i has at most $d_i - 1$ processors that can write to it. Combining this with Observation 2.1 we get

$$M_i \leq M_{i-1} + (d_i - 1)N_{i-1}.$$

Since the number of states of a processor just before the read of step i is at most Z_{i-1}, there are at most Z_{i-1} possibilities for the cell that a processor reads in step i. Along with Observation 2.1, this gives

$$N_i \leq N_{i-1} + Z_{i-1}M_i.$$

Since $M_0 = 1$ and $N_0 = 0$, the stated bounds may be proved by induction.

If the additional conditions are satisfied,it is easy to show, by induction, that for $i \geq 1$,

$$2^i \Delta_i \leq d_i^2 \tag{1}$$

which proves the stated bounds. ∎

3.1 Making a Computation Graph Regularized

In this section we show how to regularize a computation graph by setting a small number of inputs. The idea is to fix all the vertices at level i, for $i = 1, 2, \ldots$, that have indegree at least d_i. When we have done this for levels 1 through $i-1$, the computation graph is S-regularized upto level $i-1$. Then by Lemma 3.1 the number of input variables that can affect a processor at level i is small. Lemma 3.2 shows that by appropriately setting the variables that affect a processor, we can fix a processor to any desired state. Let p be the highest priority processor that can write to a cell c at level i. If we fix p to the state in which it writes to c, then, since all other processors that may write to c have a lower priority, c will always have the contents written by p, and will therefore be fixed.

In this fashion, we may fix every vertex at level i that has indegree at least i. In Lemma 3.3 we show that the total number of input variables set is small.

Lemma 3.2 *Let b be a partial input and let $x \in D^n$ be such that $x \leq b$. Let G be a computation graph of any algorithm that takes inputs from D^n and consider $G(b)$. Let p be a processor and c a cell. Let b' and b'' be partial inputs, $x \leq b', b'' \leq b$ such that in b', no variable in* affect$(b,(p,i))$ *has value $*$, and in b'', no variable in* affect$(b,(c,i))$ *has value $*$. Then* states$(b',(p,i))$ = state$(x,(p,i))$. *Similarly,* contents$(b'',(c,i))$ = contents$(x,(c,i))$.

Proof. If $\forall x' \leq b'$, $state(x',(p,i)) = state(x,(p,i))$ then the lemma holds, so assume $\exists x' \leq b'$ such that $state(x'(p,i)) \neq state(x,(p,i))$.

Notice that x and x' can differ only on input variables not in $affect(b,(p,i))$, since $x, x' \leq b$ and in b' each variable in $affect(b,(p,i))$ is set to a value in D. Let r be the number of variables on which x and x' differ. Let $y_0 = x, y_1, \ldots, y_r = x'$ be inputs such that for each i, $1 \leq i \leq r$, $y_i \leq b$ and y_i differs from y_{i-1} exactly on one variable. Since $state(y_0,(p,i)) \neq state(y_r,(p,i))$, $\exists j$ such that $state(y_{j-1}(p,i)) \neq state(y_j,(p,i))$. Let x_j be the variable on which y_{j-1} and y_j differ. Then, by definition, $x_j \in affect(b,(p,i))$, a contradiction.

A similar proof holds for a cell and its contents on b'' ∎

Lemma 3.3 *Let $d_0 = m \geq 4$ and $d_{i+1} = d_i^6$, for $i \geq 0$. Define $S = (d_0, d_1, d_2, \ldots)$. Let G be a computation graph of an algorithm that uses P processors and takes inputs from D^n, where $|D| \leq m$. Then there exists a S-regularizing input in which at most P/m variables do not have value $*$.*

Proof. We describe a simple procedure to find such a S-regularizing input. Our strategy is to proceed level by level, refining the current partial input at each level. When we are finished with level i, the current partial input will be such that at levels $j \leq i$, all vertices of indegree $\geq d_j$ will be fixed.

Suppose we have finished with levels $1, \ldots, i-1$, and are currently at level i. Let b denote the current partial input. Consider the computation graph on b and let (c,i) be a free vertex of indegree $\geq d_i$. Let p be the highest priority processor that could write to (c,i). Then $\exists x \in D^n$, $x \leq b$, an input on which p writes to (c,i). Let b' be the partial input obtained from b by setting each input variable in $affect(b,(p,i-1))$ consistently with x. By Lemma 3.2, on any partial input $b'' \leq b$, $states(b'',(p,i-1)) = \{state(x,(p,i))\}$. Hence, on all inputs consistent with b', p will write the same value to (c,i), so this vertex is fixed. We set the current input to be b' and repeat the process.

Since we are continuously refining the input, the degree of a vertex cannot increase. Thus, the procedure will eventually fix all the vertices in level i with indegree $\geq d_i$.

It remains to bound the number of input variables set. When we are at level i, the current input is such that all free vertices at levels $j \leq i$ have indegree $\leq d_j - 1$. By definition, $|affect(b,(p,i-1))| \leq N_{i-1}$.

Hence, to fix each vertex, we set at most N_{i-1} variables. By definition, each processor writing to a cell at level i may be in at most Z_{i-1} states, and hence may contribute at most this many edges to the graph. Thus, the number of edges between levels $i-1$ and i is at most $Z_{i-1}P$, implying that the number of vertices with indegree $\geq d_i$ is at most $Z_{i-1}P/d_i$. At level i, therefore, at most $N_{i-1}Z_{i-1}P/d_i$ input variables are set to values in D. By Lemma 3.1 this is at most $d_{i-1}^4 P/d_i \leq P/d_{i-1}^2 \leq P/2^{i+1}m$, where the last inequality holds because for each $i \geq 0$, $d_i \geq m$ and $d_i \geq 2^{i+1}$. Summing for all i, we get the bound on the total number of variables set. ∎

4 Everywhere Sensitive and Elusive Boolean Functions

We give an alternate, but equivalent definition of everywhere sensitivity.

Recall from Section 2.2 that a partial input $b \in D \cup \{*\}^n$ and $X(b)$ is the set of all inputs consistent with b. Define the *length* of a partial input, written $|b|$ to be the number of values in it that are not $*$'s. For a function $f : D^n \longrightarrow R$, and a partial input b, let $R(b)$ denote the set of possible output values on inputs in $X(b)$. That is, $R(b) = \{r : r \in R \text{ and } \exists x \in X(b) \text{ such that } f(x) = r\}$.

Then we define the everywhere sensitivity of f to be $\max\{k : \forall \text{ partial inputs } b,\ |b| \leq k \Longrightarrow |R(b)| > 1\}$. It may be verified that this definition is equivalent to the one in Section 1. Thus we may view the everywhere sensitivity of a function as the maximum number of input variables that an adversary may reveal, without revealing the value of the function. This is precisely the view that we will use in our proofs. We now obtain a lower bound through the following simple argument.

Theorem 4.1 *Let $f : D^n \longrightarrow R$ be a function with everywhere sensitivity* $\mathrm{es}(f)$. *Let $k = \lceil \frac{1}{4}\log(\frac{\log \mathrm{es}(f)}{\max\{1, \log 4P|D| - \log \mathrm{es}(f)\}}) \rceil$. Then, any CRCW PRAM algorithm computing f with P processors requires k steps.*

Proof. Assume that $P \geq n$. The function cannot be computed faster by using less processors, hence the lower bound for $P = n$ also holds for $P < n$.

Choose $m = 4P/es(f)$ so that $P/m \leq es(f)/4$. Since $es(f) \leq n$ and $P \geq n$, note that $m \geq 4$. Let $d_0 = m$, $d_{i+1} = d_i^6$, $i \geq 0$, and define $S = (d_0, d_1, d_2, \ldots, d_k)$. The choice of k in the theorem ensures that $d_k^2 < es(f)/2$.

Suppose there is an algorithm that computes the function in less than k steps. Consider the computation graph of the algorithm. By Lemma 3.3 there is an S-regularizing input with at most P/m variables set to values in D. Let b be this partial input and consider the output cell of the computation graph, (β, k). By Lemma 3.3, $|affect(b, (\beta, k))| \leq d_k^2$. Choose any input $x \in D^n$, $x \leq b$. Let b' be the input obtained from b by setting all the variables in $affect(b, (\beta, k))$ consistently with x. By Lemma 3.2, on any input $x' \in D^n$, $x' \leq b'$, the algorithm outputs the same value, that is, $content(x', (\beta, i))$.

However, since at most $P/m + d_k^2 \leq es(f)/4 + es(f)/2 < es(f)$ variables in b' are set to values in D, there must be two inputs consistent with b' on which the function has different values. This gives us a contradiction. ∎

We now introduce a measure that allows us to quantify the complexity of a function more accurately. Everywhere sensitivity is more robust than sensitivity in that its value is unaffected by small numbers of inputs. However, it errs in the other direction, that is, it is often insensitive to large numbers of variables. Consider the Boolean function $f(x_1, \ldots, x_n) = x_1 \vee (\bar{x_1} \wedge PARITY(x_2, \ldots, x_n))$. It is easy to see that $es(f) = 0$. Clearly this function has a low everywhere sensitivity, but is hard to compute. This motivates our definition of another measure.

For a partial input b and a function $f : D^n \longrightarrow R$, $f|_b$ is the function obtained by replacing input variable x_i with the value assigned to it by b, where a value $*$ indicates that the variable may assume any value in D.

The *elusiveness* of a function f, written $E(f)$ is $\max\{es(f|_b) : b \text{ is a partial input}\}$. Clearly, $E(f) \geq es(f)$. On the other hand, the difference between the two may be arbitrarily large. This is demonstrated by the function above, which has everywhere sensitivity 0 and elusiveness $n-2$.

We may now strengthen the above theorem as follows. If f is a function of elusiveness $E(f)$, then there is a restriction σ such that $es(f|_\sigma) = E(f)$. Applying Theorem 4.1 now yields

Theorem 4.2 *Let $f : D \longrightarrow R$ be a function. Let $k = \lceil \frac{1}{4}\log(\frac{\log E(f)}{\max\{1, \log 4P|D| - \log E(f)\}}) \rceil$. Then, any CRCW PRAM algorithm computing f with P processors requires k steps.*

5 Applications

The problems of approximate selection and approximate counting are:
Approximate Selection: Given n elements from an ordered universe, an integer $r \in \{1, \ldots, n\}$ and an accuracy parameter $\lambda \geq 1/(n+1)$, find an element with rank between $r - \lambda n$ and $r + \lambda n$.
Approximate Counting: Given an input from $\{0,1\}^n$, and an accuracy parameter $\lambda \geq 1/(n+1)$ compute an integer b, $s/(\lambda+1) \leq b \leq s(\lambda+1)$, where s is the number of 1's in the input.

Theorem 5.1 *Approximate selection problems with accuracy $\lambda \leq 1/4$ using Cn processors requires time*

$$\Omega(\log \frac{\log n}{\max\{1, \log C\}})$$

Proof. We assume that $C \geq 2$ and $\lambda = 1/4$; clearly the problem cannot be solved faster by using less processors or by solving for smaller λ.

In order to prove a lower bound, we consider a restricted version of the problem where each element has a value in $\{0,1\}$ and the problem is to approximately select the median. Any deterministic algorithm that solves this problem with accuracy λ can be viewed as computing a Boolean function $f : \{0,1\}^n \longrightarrow \{0,1\}$.

We will show that the function f has everywhere sensitivity at least $n/2 - \lambda n - 1$.Consider any partial input b of length less than $n/2 - \lambda n$. Let b_0 and b_1 be elements of $X(b)$ obtained by setting all the $*$'s in b to 0 and 1 respectively. Clearly, $f(b_0) = 0$ and $f(b_1) = 1$. Thus $es(f) \geq n/2 - \lambda n - 1 \geq \frac{n}{2}(\frac{1}{2} - \lambda)$. Using Theorem 4.1 with $P = Cn$ and $\lambda = 1/4$ gives us the claimed bound. ∎

Theorem 5.2 *Approximate counting with accuracy λ using Cn processors requires time*

$$\Omega(\log \frac{\log n}{\max\{1, \log\lambda + \log C\}}).$$

Proof. We prove the bound for $\lambda, k \geq 2$; clearly, the problem cannot be solved faster by using less processors or by solving for a smaller λ.

Any deterministic algorithm that solves the approximate counting problem can be viewed as computing a function $g : \{0,1\}^n \longrightarrow \{0, \ldots, \lceil \lambda n \rceil\}$, where, for $x \in \{0,1\}^n$, $g(x)$ is the value output by the algorithm. Note that if x has s 1's, $s/2\lambda \leq g(x) \leq 2s\lambda$.

We will show that the function g has everywhere sensitivity at least $n/2\lambda^3$. Consider any partial input b, of length less than $n/2\lambda^3$. Let $b_0 \in X(b)$ be the partial input obtained from b by setting all the $*$'s in b to 0, and let $b_1 \in X(b)$ be the input obtained by setting all the $*$'s to 1. Clearly, $g(b_0) < n/\lambda^2$, since b_0 has less than $n/2\lambda^3$ 1's. Similarly, $g(b_1) \geq 3n/8\lambda$, since b_1 has at least $3n/4$ 1's. For sufficiently large n, $g(b_0) \neq g(b_1)$, so $es(g) \geq n/2\lambda^2$. Applying Theorem 4.1 with $P = Cn$ yields the stated bound. ∎

5.1 Upper Bounds

Our lower bounds hold for nonuniform algorithms. In this section we show that for nonuniform algorithms, they are the best possible. Consider the problem of approximate counting with $\lambda = 2$. It is known that there is an algorithm that solves this problem in time $\mathcal{O}(\log\log n)$ using n processors [8]. Since this is equivalent to computing a function of everywhere sensitivity at least $n/16$, Theorem 4.1 gives a tight lower bound of $\Omega(\log\log n)$ bound. In contrast, the best uniform algorithm for this problem takes $\mathcal{O}((\log\log n)^3)$ time. Closing the gap between the two bounds remains an open question.

References

[1] Y. Azar. Lower bounds for Threshold and Symmetric Functions in Parallel Computation. *SIAM Journal on Computing*, Vol. 21, No. 2, (1992), pp. 329-338.

[2] P. Beame. Lower Bounds in Parallel Machine Computation. *Ph.D. Thesis*, University of Toronto, (1991).

[3] P. Beame and J. T. Håstad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM*, **36** (1989), pp. 643–670.

[4] S. Chaudhuri. A Lower Bound for Linear Approximate Compaction. In *Proc. 2nd Israel Symp. on Theory of Comp. and Sys.*, (1993), pp. 25-32.

[5] S. Chaudhuri, T. Hagerup and R. Raman. Approximate and Exact Deterministic Parallel Selection. In *Proc. 18th Math. Fdtns. of Comp. Sci.*, (1993), *to appear.*

[6] S. Cook, C. Dwork and R. Reischuk. Upper and Lower Time Bounds for Parallel Random Access Machines Without Simultaneous Writes. *SIAM*

Jouran on Computing, Vol. 15, No. 1, (1986), pp. 87-97.

[7] T. Hagerup. Fast Deterministic Processor Allocation. In *Proc. 4th ACM-SIAM SODA* (1993), pp. 1-10.

[8] T. Hagerup. *personal communication.*

[9] T. Hagerup and R. Raman. Waste makes haste: Tight bounds for loose parallel sorting. In *Proc. 33rd IEEE FOCS* (1992), pp. 628–637.

[10] T. Hagerup and R. Raman. Fast Approximate and Exact Parallel Sorting. In *Proc. 5th Annual SPAA* (1993), pp. 346-355.

[11] J. Håstad. *personal communication.*

[12] J. JáJá. An Introduction to Parallel Algorithms. Addison-Wesley, Reading, Mass., 1992.

[13] Y. Matias and U. Vishkin. Converting High Probability into Nearly-Constant Time - with Applications to Parallel Hashing. *In* Proc. 23rd Annual STOC, (1991), pp. 307-316.

[14] N. Nisan. CREW PRAMs and Decision Trees. *SIAM Journal on Computing*, Vol. 20 (1991).

[15] H-U. Simon. A tight $\Omega(\log\log n)$ bound on the time for parallel RAM's to compute nondegenerated Boolean functions. In *M. Karpinski, ed., Foundations of Computing Theory*, Lecture notes in Comput. Sci., 158 (Springer, Berlin) pp. 439-444.

[16] G. Turan. The Critical Complexity of Graph Properties. *Information Processing Letters*, 18, (1984) pp. 151-153.

[17] U. Vishkin and A. Wigderson. Trade-offs between depth and width in parallel computation. *SIAM Journal on Computing.*, 14 (1985) pp. 303-314.