ソフトウェア基礎論　23－10
（1987. 12. 10）

同時書込みを許すＰＲＡＭの計算時間の下限について

Very Small Tight Bounds on the Time of Uniform PRAMs with Simultaneous Writes

岩間一雄

Kazuo Iwama

京都産業大学

Kyoto Sangyo University

あらまし．　A(i,j) をアッカーマン関数とし，$\overline{A}_k(n)$ を$\overline{A}_k(n)$ ＝（A(i,j) ≧ n　である最小の j）で定義される（A(i,j) の逆）関数とする．同時書き込みを許し，演算として＋，－，ビットごとのAND，ビットごとのORの４種を許す並列乱アクセス機械（ＰＲＡＭ）の計算時間の下限に関し，つぎの定理を証明する：任意の c≧４に対し，このようなＰＲＡＭ（多項式プロセッサ数）によりΘ($\overline{A}_c$(n))ステップで計算される非退化のｎ変数論理関数が存在する．

Let $A(i,j)$ be the Ackermann function and let $\overline{A}_k(n)$ be its inverse function defined by $\overline{A}_k(n)$ = least j such that $A(k,j) \geq n$. We prove very small, nonconstant, tight upper (lower) bounds for the computation time of uniform PRAMs with concurrent writes and with operations +, −, bitwise OR and bitwise AND: For any constant $c \geq 4$, there is a nondegenerate Boolean function G_c of n variables such that it takes $\Theta(\overline{A}_c(n))$ steps to compute G_c by such PRAMs with polynomial number of processors.

1. Introduction.

Let $A(i,j)$ be the Ackermann function and let $\overline{A}_k(n)$ be its inverse function defined by $\overline{A}_k(n)$ = least j such that $A(k,j) \geq n$. CRCW-PRAMs denote parallel RAMs with simultaneous writes and with operations +, −, & (bitwise AND) and | (bitwise OR). It should be noted explicitly that CRCW-PRAMs in this paper are *uniform*, namely, each RAM has the same program not depending on the size of inputs. We prove very small, nonconstant, tight upper (lower) bounds for the computation time of such CRCW-PRAMs:

Main Theorem. For any constant $c \geq 4$, there is a nondegenerate Boolean function G_c of n variables such that it takes $\Theta(\overline{A}_c(n))$ steps to compute G_c by CRCW-PRAMs with polynomial number of processors.

The parallel random access machines (PRAMs) have clearly become a standard model for parallel computation, over which a huge number of fast algorithms were developed. It seems, however, that studies on the model itself have drawn interests of much less researchers and there still remain a lot of unknowns. For instance:

(1) For PRAMs *without* simultaneous writes (CREW-PRAMs), one can show[8] that $\Omega(\log\log n)$ is the general lower bound of the time to compute nondegenerate Boolean functions and at the same time it is an upper bound for a number of specific functions [7]. As for CRCW-PRAMs, however, although it is known that some simple functions have interesting lower bounds like $\Omega(\log n/\log\log n)$ for the PARITY function[5, 10] and that trivial functions like n-bit OR can be computed in a constant time, almost nothing was known about the large gap between them.

(2) Note that the uniformity is the unique feature that distinguishes CRCW-PRAMs from unbounded fan-in circuits [9] and, in the author's opinion, it is this uniformity that makes CRCW-PRAMs an acceptable parallel model. However, it is again very difficult to find literature explicitly investigating this feature. (Many authors of recent papers [1, 2, 3, 4, 5, 6, 7, 10] succeeded in achieving their nontrivial results, lower bounds in most cases, without assuming the uniformity. That is nothing but special cases. There seems to be a lot of cases in which removing the uniformity makes problems trivial, in other words, the uniformity gives us interesting questions. Consider, for example, the following Boolean function. $f(x_1, \cdots, x_n)=1$ iff exactly three variables are 1 and $x_i=x_j=x_k=1$ implies $i=\gcd(j,k)$. It is obvious that f is computed by constant depth unbounded fan-in circuits (or nonuniform CRCW-PRAMs) of polynomial circuit size but it is unlikely that f can be computed by uniform CRCW-PRAMs with a normal instruction set (such as +, −, &, |, ×, ÷, etc.) in constant steps. Also for the PARITY function recently popular [1, 5, 10], it is not known, to the best of the author's knowledge, if we

can compute n-bit PARITY in constant steps by uniform CRCW-PRAMs of unbounded number of processors.)

Our present result gives us a new knowledge about those (1) and (2), that includes: (i) Although it is still open if there is a general lower bound like that for CREW-PRAMs (e.g., we cannot presently replace $\overline{A}_c(n)$ by $\overline{A}_n(n)$), the main theorem shows it must be very small if exists. Note that the result fully depends on the uniformity, in other words, such lower bounds do not exist trivially for nonuniform CRCW-PRAMs. (ii) In spite of the restricting uniformity imposed, CRCW-PRAMs still exhibit surprisingly high performance for a certain type of functions. As shown later, they can compute, in a sense, $\overline{A}_c(n)$ from n in constant steps with the aid of input forms and the bitwise operations & and $|$. (It is another interesting open question whether we can achieve similar upper bounds without & or $|$.) The proof includes several tricky ideas.

2. The Model.

The following machine is a minimal structure necessary to prove the upper bound of the theorem. In this paper we define the machine as the one which computes a mapping from Σ^n to Σ for a finite set Σ of symbols. A CRCW-PRAM consists of processors $P_0, P_1, \cdots$, all of which hold the same program. In the program we can use a finite number of local variables $x, y, z, \cdots$, a common array $M[0], M[1], \cdots$, and the following instructions.

```
$x \leftarrow$ constant (an integer or a character in $\Sigma$)
$x \leftarrow$ processor number
$x \leftarrow y \cdot z$,   $\cdot \in \{+,-,\&,|\}$ ($x$ and $y$
               must hold integers)
$M[x] \leftarrow y$
$y \leftarrow M[x]$
GOTO label if $x=y$ ($x$ and $y$
               may hold characters)
HALT
```

If more than one processor attempts to write into the same element of the common array, the lowest numbered processor succeeds (PRIORITY). An input of size n is a string $a_0 a_1 \cdots a_{n-1}$, each $a_i \in \Sigma$ is placed in $M[i]$. At time zero all the processors, the number of which is limited by some polynomial $p(n)$, begin their computation. When the HALT instruction is executed, it must be done at the same time by all those processors. The answer (a symbol in Σ) is placed in $M[0]$. (See e.g., [9] for more detailed general description of PRAMs.)

Remarks. (i) $x \leftarrow$ processor number is especially important. That is only one possibility for each processor to recognize the difference of itself from other processors. (ii) Σ may contain more than two symbols. Our machine may be viewed as a recognizer of languages over Σ. The theorem holds for any Σ including two or more (but of course finite) symbols. (iii) $\{\&,|\}$ may be replaced by $\{\times,\div\}$ or $\{$ mod $\}$ and the proof becomes easier. Note that bit-wise COMPLEMENT, which can create a large number in a single step, is excluded. O(log$n$) bits are enough for each variable. (iv) As for the lower bound, we do not need the uniformity. It holds for the circuit model. (v) The theorem is still true for the COMMON resolution (simultaneous writes must be done for the same value).

3. Proof of the Theorem.

First we will sketch roughly what the function G_c of the theorem looks like and why that is desirable to our goal. For simplicity the constant c is fixed to 5 and the generalization will be mentioned at the end. Note that $\overline{A}_5(n)=k$ means $\log^*\log^* \cdots \log^* n$ (k times) becomes 1 but not by $k-1$ times and $\log^* n=k$ means $\log\log \cdots \log n$ (k times) becomes 1 but not by $k-1$ times.

What we want to do is as follows. It is known [5, 10] that depth k PARITY circuits of m variables need $\Omega(2^{m^{1/4k}})$ gates. So, if $m=(\log n)^{g(n)}$ then we need $\Omega(g(n))$ depth to make the number of gates polynomial. As $g(n)$ we now take $\overline{A}_5(n)$. Now consider language

$$L=\{\sigma=\sigma_1\sigma_2 \mid |\sigma|=n,\ |\sigma_2|=(\log n)^{\overline{A}_5(n)}$$

$$\text{and } \sigma_2 \text{ contains odd number of 1's}\}.$$

By the above known fact, to recognize L needs $\Omega(\overline{A}_5(n))$ depth even by circuits, which achieves our lower bound. Therefore we are done if we can achieve the upper bound or if we (i) can compute $\overline{A}_5(n)$ from σ to determine the small portion σ_2 of the whole σ and (ii) can compute PARITY of that σ_2, both very quickly (at most $\overline{A}_5(n)$ steps). Note that those (i) and (ii) are trivial jobs for circuits or nonuniform CRCW-PRAMs, but not at all for uniform CRCW-PRAMs. As for (ii), for instance, it is very unlikely to be able to compute PARITY of length $(\log n)^{\overline{A}_5(n)}$ in $\overline{A}_5(n)$ steps. (The situation should be recognized correctly. Imagine trying to compute $\log n$-bit PARITY. In the circuit model, we can split all $2^{\log n}=n$ different bit-patterns using $O(n)$ gates and can select desirable (having odd number of 1's) patterns by "us" or by e.g, slowly moving TM's having nothing to do with the model involved. Namely how to do this selection is not included in the algorithm. In the present situation, this selection is included in the program, in other words, it must be done by the uniform CRCW-PRAM program quickly.)

We first construct the following language L_5 over alphabet Σ. Σ includes 0, 1, #, a, b, c, d and 0, 1, # with mark ', ", or $^@$. (More than one mark may be placed like $0'^@$.) In order for a string σ_n to be in L_5, first of all, it has to consist of five portions:

$$\sigma_n = \alpha\beta\gamma\delta\lambda$$

Note that n of σ_n somehow shows the size but not the exact length. The length of σ_n is about n^4 and as one can see later n must be 2^l for some l. Among the five portions of σ_n, α is determined to make possible the quick computing of $\overline{A}_5(n)$. γ acts as σ_2 of L above. However we do not try to compute PARITY of γ directly but use the following trick.

β is of the form

$$\beta = dp_1p_2 \cdots p_{n-1} \quad (p_i \in \{0,1\})$$

and we consider this string defines a function f : $\{0,1\}^{\log n} \to \{0,1\}$ as $f(0,\cdots,0)=0$ and $f(x_1,\cdots,x_{\log n}) = p_{\mathrm{bin}(x_{\log n}\cdots x_1)}$. γ must be of the form

$$\gamma = dq_1q_2\cdots q_h\#\#\cdots\#$$
$$(q_i \in \{0,1\},\ |\gamma|=n,\ h=(\log n)^{\overline{A}_5(n)}).$$

This is a key portion to determine whether σ_n is in L_5, namely it is so iff $f^h(q_1,q_2,\cdots,q_h)=1$, where f^h is the usual composition of the above f. δ and λ are supplementary portions necessary to compute f quickly, which we will mention later. The key lemma in this paper is (proof is given later):

Lemma 1. L_5 can be recognized by a CRCW-PRAM in $O(\overline{A}_5(n))$ steps.

The next step is an easy extension. Let L_5^{B} be the language over $\{0,1\}$ obtained by the usual coding method from L_5. (0 and 1 should be mapped to be adjacent like 0 $\to$ 1110 and 1 $\to$ 1111.) The next lemma is almost a corollary of Lemma 1.

Lemma 2. L_5^{B} can be recognized by a CRCW-PRAM in $O(\overline{A}_5(n))$ steps.

G_5 is the Boolean-function version of this L_5^{B}. Now we prove the lower bound.

Lemma 3. If L_5^{B} can be recognized by a CRCW-PRAM in $d(n)$ steps then the PARITY function of $(\log n)^{\overline{A}_5(n)}$ variables can be computed by unbounded fan-in circuits of polynomial (on n) size in depth $O(d(n))$.

To prove this lemma we first construct the circuit of depth $O(d(n))$ that computes L_5^{B} by the way of [5, 10] and then fix the values of variables corresponding to α, β, δ, λ and other than the key bits of γ (recall that 0 and 1 were mapped to be the same but exactly one bit). As one can see in a moment, α, δ and γ are, for σ_n to be in L_5, completely determined only by the integer n. As for β, we have to fix it so as that f computes the PARITY function. Now [5, 10] and simple calculation leads us to:

Lemma 4. The circuits (and therefore CRCW-PRAMs by [9]) need $\Omega(\overline{A}_5(n))$ steps to recognize L_5^{B}.

Proof of Lemma 1. What we have to do is to describe the conditions for the strings α, β, γ, δ and λ to meet and to show that CRCW-PRAMs can check the conditions quickly. We shall first take α. When n=16, α looks like

c 0000###########*a* 1′000###########

a 01′00###########*a* 110′0########### $\cdots$

a 0111##########′#*a* 1111###########′

b 0″0## $\cdots$ *a* 1′0## $\cdots$ *a* 01′## $\cdots$ *a* 11#′# $\cdots$

a####′# $\cdots$ $\cdots$

b 0#″## $\cdots$ *a* 1′### $\cdots$ *a*##′## $\cdots$

b###″# $\cdots$ *a*#′## $\cdots$

$\cdots$ $\cdots$

c 0@ 0## $\cdots$ *a* 1′0## $\cdots$ *a* 01′## $\cdots$ *a*###′# $\cdots$

$\cdots$ $\cdots$

More formally

$$\alpha=\alpha_{0,0,0}\alpha_{0,0,1}\cdots\alpha_{0,0,n-1}\alpha_{0,1,0}\cdots\alpha_{0,1,n-1}\cdots$$
$$\alpha_{0,n-1,0}\cdots\alpha_{0,n-1,n-1}\alpha_{1,1,0}\cdots\alpha_{n-1,n-1,n-1}$$

The rule is as follows:

(1) $|\alpha_{i,j,k}|=n$ for all $0 \le i,j,k \le n-1$. Hence $|\alpha|=n^4$. We make a group of $\alpha_{i,j,0}$, $\alpha_{i,j,1}$, $\cdots$, $\alpha_{i,j,n-1}$, which we call (i,j)-group.

(2) If $k \ne 0$ then $\alpha_{i,j,k}$ begins with a, followed by $n-1$ symbols of 0, 1 or # and has mark ′ on the kth symbol. Note that if $k=n-1$ then the mark is placed on the last symbol of $\alpha_{i,j,k}$.

(3) If $j \ne 0$ and $k=0$ ($j=k=0$, respectively), $\alpha_{i,j,k}$ begins with b (with c) and has mark ″ on the jth (@ on the ith) symbol. As before those marks are placed on the rightmost symbol when $j=n-1$ or $i=n-1$.

(4) If we ignore the marks, $\alpha_{i,j,k}$ can be written as

$$\alpha_{i,j,k}=uN(i,j,k)\#\#\cdots\#$$

where $u \in \{a,b,c\}$, and $N(i,j,k)$ is the binary representation of integer k (we will simply say "$N(i,j,k)$ is k") or ## $\cdots$ # (we will say "$N(i,j,k)$ is #'s"). Hereafter the marks will often be ignored like this when they are not important. $N(i_1,j_1,k_1)$ and $N(i_2,j_2,k_2)$ have the same number of digits if $i_1=i_2$ and $j_1=j_2$, which must be minimum enough to hold the maximum integer in that (i_1,j_1)-group. (See above example. $|N(0,1,0)|=2$ since the maximum integer in that group is 3.)

(5) $N(0,0,k)=k$ for all k and $N(0,0,n-1)=11\cdots1$. Thus n must be 2^l for some l.

(6) $N(0,1,k)=k$ if $k<$ the number of digits of $N(0,0,0)$. In general $N(0,j,k)$ is k if $|N(0,j-1,0)| \ge 2$ and $k<|N(0,j-1,0)|$. If $|N(0,j-1,0)| \le 1$ then $N(0,j,k)$ is #'s for all k.

(7) Let $j_0=\mathrm{MIN}\{j \mid N(0,j,0)=\#'s\}$, namely, $N(0,j_0-1,0)$ is not #'s and $N(0,j_0,0)$ is #'s. Then $N(1,0,k)=k$ if $k<j_0$. In general $N(i,0,k)=k$ if $j_0=\mathrm{MIN}\{j \mid N(i-1,j,0)=\#'s\} \ge 2$ and $k<j_0$. $N(i,j,k)$ ($j \ne 0$) is determined by the similar rule as (6).

α might look complicated, but the idea is simple, namely, we used the fact that $\lceil \log n \rceil$ = the number of digits of the binary representation of n. Now we claim the following two facts to show such α is desirable.

Claim 1. If α satisfies the rule above then $\bar{A}_5(n) = \mathrm{MIN}\{i \mid N(i,0,0)=\#'s\}$.

Proof. Obvious by the rule above and by the definition of $\bar{A}_5(n)$.

Claim 2. CRCW-PRAMs can check if α satisfies the rule in constant steps.

Proof. The following description of the algorithm may be a little informal but would be better for the readability. We assign one processor P_i per each input symbol held in $M[i]$. At the very beginning each P_i reads $M[i]$ using its own processor number and save the input symbol into some local variable. After that $M[i]$ is mostly used for the purpose of communication between processors. If there is a case when each processor P_i needs to read the input symbol originally held in $M[j]$ $(i \neq j)$, the saved input symbols are reloaded to the common array at the preceding step.

Step 1. Remember that we have to do everything in constant steps. In the first step, we verify symbols a, b and c are placed at the same intervals. Each processor introduces four variables, say, v_n, v_{n^2}, v_{n^3} and v_{n^4}. Now all the processors reading input symbol a (i.e., those which saved symbol a at the beginning) write their processor number to $M[0]$. Thanks to the PRIORITY resolution of the simultaneous writes, we can get the position of the leftmost a in $M[0]$. All the processors then read $M[0]$ and save the value into v_n. Similarly for v_{n^2} (the position of the leftmost b), v_{n^3} (the second leftmost c) and v_{n^4} (the leftmost d that is the first symbol of β). Next all the processors P_i reading input symbol a or b or c (i.e., not reading 0 or 1 or #) send a "signal" to P_{i+v_n} (write e.g., some special integer to $M[i+v_n]$). Then all the processors P_j check whether the following two conditions are met: (i) The first processor P_0 is reading c. (ii) If P_j received the signal then it is reading a or b or c. (iii) If P did not receive the signal then it is NOT reading a or b or c (i.e., is reading 0 or 1 or #). To check (ii), all the processors that received the signal and is reading 0 or 1 or #, write, say, 1 into $M[0]$ which has to be cleared in advance. Then by reading $M[0]$ all the processors can tell whether condition (i) is met. If either of the conditions is not met then all the processors halt and reject the input. This basic technique to check in constant steps if some symbol appears in the same interval, is used frequently below also. Similar check is done for the intervals of b and c using v_{n^2} and for the intervals of c using v_{n^3}.

As before we make grouping of the processors. We define (i,j,k)-group ((i,j)-group and (i)-group, respectively) as a set of n (n^2 and n^3, respectively) processors which are reading the symbols of $\alpha_{i,j,k}$ ($\alpha_{i,j,0} \cdots \alpha_{i,j,n-1}$ and $\alpha_{i,0,0} \cdots \alpha_{i,n-1,n-1}$, respectively). (i,j,k)-groups, (i,j)-groups and (i)-group are sometimes called s-group, m-group and l-group, respectively.

Step 2. Only the leftmost m-group ((0,0)-group) of processors check whether the marks ' are placed on proper symbols of $\alpha_{0,0,0}$ to $\alpha_{0,0,n-1}$. (Other processors whose processor number $\geq v_{n^2}$ sleep during this step.) (i) The leftmost processor P_0 that is reading c of $\alpha_{0,0,0}$ and each processor P_i reading a symbol marked by ' sends a signal to P_{i+v_n+1}. (ii) All the processors which received the signal check if they are reading the marked symbol. (iii) All the processors but those of above (ii) must not be reading the marked symbol. (iv) $P_{v_{n^2}-1}$ must be reading the marked symbol. If this test is passed then it is guaranteed that $v_{n^2}=v_n \times v_n$.

Step 3. Do the same thing as Step 2 for the leftmost l-group of processors and the mark ". $v_{n^3}=v_{n^2} \times v_n$ is guaranteed.

Step 4. Do the same thing as Step 2 for the entire α and the mark $^{@}$.

Step 5. Check the position of mark ' for the whole string α. Each processor P_i reading the marked symbol sends a signal to $P_{i+v_{n^2}}$.

Step 6. Do the same as Step 5 for the mark ". P_i sends a signal to $P_{i+v_{n^3}}$.

Step 7. Only the leftmost m-group of processors check whether $N(0,0,0) \cdots N(0,0,n-1)$ are proper. (i) Each s-group of processors determine their representative (the processor of the least processor number within the group). To do so, each processor calculates

(its own processor number $\mid (v_n-1))-(v_n-1)$,

that is the processor number of the representative we wanted. (ii) Check whether $|N(0,0,0)| = |N(0,0,1)| = \cdots = |N(0,0,n-1)|$. Within each s-group, processors can know the leftmost # by the simultaneous writes to the representative. Thus all representatives hold the length of the binary number of their group and then they send the value to the next representative of each. If the value sent is equal to the value of its own in all the representatives then the test is passed. (iii) Check if $N(0,0,0)=00 \cdots 0$ and $N(0,0,n-1)=11 \cdots 1$. (iv) Each s-group calculates the binary number $N(0,0,k)$ of that group plus one and sends its value to its next s-group. To do so, processors in each s-group first find the leftmost 0 within that s-group. Then each processor (reading 1) placed to the left of that 0 sends 0 to the processor of the same position of the next s-group. Similarly, the processor reading the leftmost 0 sends 1 and each processor to the right of the leftmost 0 sends the same symbol as it reads. Check if the symbol sent = its own. Now it is guaranteed that $n=2^l$ for some l. The reason why this condition is needed will be given later.

Step 8. All the m-groups ((i,j)-groups) check $N(i,j,0), \cdots, N(i,j,n-1)$. What should be done is almost the same as Step 7 but the following. (i) Each m-group also has to determine its representative. Unlike Step

7, only a part of s-groups of each m-group hold binary numbers. We have to find the rightmost s-group for which $N(i,j,k)\neq$#'s. Although the rightmost binary number in each m-group does not have to be $11 \cdots 1$, its most significant bit must be 1.

So far we have verified that the binary numbers in each m-group are increasing properly. What we have to do next is to check the number of digits of those binary numbers.

Step 9. Each m-group that is reading b (not c) at its leftmost position checks whether the number of digits of the binary numbers is correct. (i) The representative of each m-group knows the number of digits of the binary numbers of its own group. (It may be 0, i.e., only #'s may appear.) (ii) The representative knows the leftmost s-group in its own m-group in which only #'s appear. (How to do is not so difficult: Each processor reading 0 or 1 writes 1 to its s-group representative. Then each representative to which no processor wrote 1, writes its processor number to its m-group representative.) Then that s-group reports the position of the ' mark to the representative. (If no ' exists then it reports 0 as the position.) (iii) The representative checks if that position reported is equal to the number of digits of the binary number of the preceding m-group. (However, if the number = 1 then it is regarded as 0.)

Step 10. Each m-group that is reading c at the leftmost position checks the number of digits. (i) Each l-group finds the leftmost m-group in its own l-group which consists of only #'s and then knows the position of mark " in that m-group. (ii) Each l-group finds the position of mark ' in the leftmost s-group consisting of only #'s in its leftmost m-group. It checks that position is equal to the position of (i).

That concludes the test whether the string α is proper. If the test is passed then we can get $\bar{A}_5(n)$ as the position of mark $^{@}$ in the leftmost l-group which contains only #'s.

Now we describe the condition strings δ and λ should satisfy. (The conditions for β and γ were already given.) We need δ as an aid to get value $i\times n$ from i quickly. It can be written as:

$$\delta=\delta_0\delta_1 \cdots \delta_{n-1}$$

where $\delta_0=d\,0^{n^2-1}$ and for $i\geq 1$ $\delta_i=b\,0^{n^2-1}$. δ_i $(i\geq 1)$ has mark ' on the ith symbol (=0) and mark " on the $(i\times n)$th symbol.

λ is needed to assign processors appropriately when computing the function f^h. It is of the form:

$$\lambda=\lambda_{0,0}\lambda_{0,1} \cdots \lambda_{0,\log n-1}\lambda_{1,0} \cdots \lambda_{1,\log n-1} \cdots \lambda_{i_0,j_0}.$$

(i) $|\lambda_{i,j}|=n$ for all i and j. (ii) $i_0\cdot\log n+j_0+1=n$, namely $|\lambda|=n^2$. (iii) $\lambda_{i,j}=u\,0^{n-1}$ where $u=d$ if $i=j=0$, $u=b$ if $j=0$ and $u=a$ if $i\neq 0$ and $j\neq 0$. (iv) In each $\lambda_{i,j}$ but $\lambda_{0,0}$ mark ' is placed on the $(i\log n+j)$th symbol and mark $^{@}$ on the jth symbol. Furthermore each $\lambda_{i,j}$ such that $j=0$ has mark " on the ith symbol. Some of them look like:

$\lambda_{0,0}=d\,0000000 \cdots\cdots 0$

$\lambda_{0,1}=a\,0'^{@}\,0000 \cdots\cdots 0$ (' and $^{@}$ on 1st)

$\cdots\ \cdots\ \cdots$

$\lambda_{0,\log n-1}=a\,0 \cdots 00'^{@}\,0 \cdots\cdots 0$ (on $(\log n-1)$th)

$\lambda_{1,0}=b\,0^{@}\,{}''\,0 \cdots 00'0 \cdots\cdots 0$ (' on $(\log n)$th)

$\cdots\ \cdots\ \cdots$

$\lambda_{2,0}=b\,0^{@}\,0''\,0 \cdots 00'0 \cdots 0$ (' on $(2\log n-1)$th)

$\cdots\ \cdots\ \cdots$

$\lambda_{i_0,j_0}=a\,0 \cdots\cdots\cdots 0'$ (' on the rightmost).

Claim 3. CRCW-PRAMs can check whether δ and λ satisfy the conditions above in constant steps.

Proof. For δ, it will be enough to point out that the interval for mark ' is n^2+1 and is n^2+n for mark ". As for λ, (i) we first set a variable, say, $v_{n\log n}$ = the position of the leftmost b (of $\lambda_{1,0}$) – the position of d (of $\lambda_{0,0}$). (ii) We verify that a's and b's are placed properly, i.e., they appear every v_n symbols. (iii) Check if the mark ' is placed in the same interval v_n+1. (iv) Check if the value of $v_{n\log n}$ is correct, by testing whether the last s-group of the first m-group (similarly as before for the definition of s- and m- groups) has the symbol marked by ' at the $(\log n-1)$st position. Recall that we already know the value of $\log n$ as the number of digits of $N(0,0,0)$. (v) Check if mark " is properly placed. The interval is $n\log n+1$. (vi) Check mark $^{@}$. This mark is placed at the same position in all the m-groups.

Claim 4. CRCW-PRAMs can compute the value of the function f^h $(h=(\log n)^{\bar{A}_5(n)})$ described before in O($\bar{A}_5(n)$) steps.

Proof We first make a preparation on string δ which helps computing $i\times n$. In each group of n^2 processors, the one that is reading mark " sends its processor number to the representative of the group. (ii) Each representative computes the value sent – the processor number of itself. Let this value be p. (iii) In each group, the processor reading mark ' knows the value p by communicating with the representative. Then it reports this value p to the processor at the same position in the first group. Thus in the first group, the ith processor holds the value $i\times n$.

Now we compute f^h using n^2 processors reading λ. Recall that each m-group consists of $n\times\log n$ processors. The first m-group is responsible to compute $f(q_1,q_2,\cdots,q_{\log n})$ where $q_1, q_2, \cdots, q_{\log n}$ are symbols (0 or 1) of γ. It might be helpful to imagine a matrix-like arrangement of processors, $\log n$ columns of n processors, each column corresponds to $\lambda_{0,j}$ and is responsible to the jth bit q_j. Note that a row of this matrix consists of $\log n$ processors whose processor numbers have the same least significant $\log n$ bits. Similarly for the second and further m-groups.

Before computing f^h, we check the number of 0 and 1 (in other words the number of #'s) in γ. (i) We find the leftmost #. (ii) The processor reading that # must be a representative of some m-group (the upper-left corner of the matrix above). (iii) That representative finds the position p of mark " in its first s-group, computes $p \times n$ using δ (see above) and sends a signal to the $(p \times n)$th (from the one reading the leftmost d of λ) processor. Namely we divide the position of the first # by $\log n$. (iv) Repeat this division exactly $\bar{A}_5(n)$ times. If the processor reading the leftmost symbol of $\lambda_{0,1}$ is sent the signal after the $\bar{A}_5(n)$th repetition, then the test is passed, namely we know γ contains exactly $(\log n)^{\bar{A}_5(n)}$ 0/1's in its left-half portion. After this test is passed, we rewrite the right-half portion of λ from #'s into 0's to make easier the job from now on. Note that those #'s are introduced to make the length of γ appropriate with keeping the function G_c nondegenerate.

To compute f^h we again repeat $\bar{A}_5(n)$ times the main loop described in a moment. In its first execution of the loop, $(\log n)^{\bar{A}_5(n)-1}$ matrixes of processors compute f for $(\log n)^{\bar{A}_5(n)-1}$ consecutive portions of γ each of which contains $\log n$ bits. The $(\log n)^{\bar{A}_5(n)-1}$ values of f computed are written back to γ by cramming from the left end. In the second execution, $(\log n)^{\bar{A}_5(n)-2}$ matixes work and so on.

In more detail, each matrix of processors do the following. Suppose that a processor P is placed at pth row and qth column of some matrix. P can tell value p as the $\log n$ least significant bits (use & operation) of its processor number and q as the position of mark @ in its s-group. (i) P wants to find whether the qth bit of its own processor number is 0 or 1. We utilize the first m-group of α as follows. (It will become clear why n must be 2^l for some l.) One can see the answer is obtained by looking at the $(p \times n+q)$th bit of α. (Again we use δ to compute $p \times n$). (ii) Now P checks if the bit obtained above is equal to the bit of γ which P is now reading. If it differs then P sends a signal to the processor at the pth row (the same as P) in the first column of the matrix. It should be noted that exactly one processor in the first column receives the signal from none of the $\log n$ processors at the same row, that means the values of the $\log n$ bits of γ to which that matrix is responsible is coincides with the values of $\log n$ least significant bits of the processor number of that row. (iii) That processor (having received no signals) looks at the pth bit of β and knows the value of the function f.

That concludes the proof of Claim 4 and also the proof of Lemma 1.

Remarks. (i) Note that the proof in this section fully depends on the fact that c of $\bar{A}_c(n)$ is a constant. Thus we cannot replace $\bar{A}_c(n)$ by $\bar{A}_n(n)$ which grows more slowly than $\bar{A}_c(n)$. (ii) The bitwise operation & is actually not necessary. It was used only when we get the $\log n$ least significant bits of a processor number. Obviously we can get that by computing the processor number − the processor number of the representative. (iii) As for the resolution of simultaneous writes we assumed the most powerful one, i.e., PRIORITY. However we can show the weakest one (COMMON) is enough. Much more processors are needed (within polynomial) but no essentially new techniques are. Details are omitted. (iv) We can use the standard technique (padding characters) to make G_c have an arbitrary number of variables.

References

1. P. Beame, "Limits on the Power of concurrent-write parallel machines," *Proc. 18th ACM Symp. on Theory of Computing*, pp. 169-176, 1986.
2. S. Cook and C. Dwork, "Bounds on the time for parallel RAM's to compute simple functions," *Proc. 14th ACM Symp. on Theory of Computing*, pp. 231-233, 1982.
3. S. Cook, C. Dwork, and R. Reischuk, "Upper and lower bounds for parallel random access machines without simultaneous writes," *SIAM J. Comput.*, vol. 15, pp. 87-97, 1986.
4. F. Fich, F. Meyer auf der Heide, P. Ragde, and A. Wigderson, "One, two, three ... infinity: Lower bounds for parallel computation," *Proc. 17th ACM Symp. on Theory of Computing*, pp. 48-58, 1985.
5. J. Hastad, "Almost optimal lower bounds for small depth circuits," *Proc. 18th ACM Symp. on Theory of Computing*, pp. 6-20, 1986.
6. M. Li and Y. Yesha, "New lower bounds for parallel computation," *Proc. 18th ACM Symp. on Theory of Computing*, pp. 177-187, 1986.
7. R. Reischuk, "Simultaneous WRITES of parallel random access machines do not help to compute simple arithmetic functions," *J. Assoc. Comput. Mach.*, vol. 34, pp. 163-178, 1987.
8. H. Simon, "A tight OMEGE (log log n)-bound on the time for parallel RAM's to compute nondegenerated Boolean functions," *Inform. Control*, vol. 55, pp. 102-107, 1982.
9. L. Stockmeyer and U. Vishkin, "Simulation of Parallel Random Access Machines by Circuits," *SIAM J. Comput.*, vol. 13, pp. 409-422, 1984.
10. A. Yao, "Separating the Polynomial-Time Hierarchy by Oracles," *Proc. 26th IEEE Symp. on Foundations of Computer Science*, pp. 1-10, 1985.

盛光印刷所