Università di Pisa

Dipartimento di Informatica

Technical Report : TR-96-25

An Insight on PRAM Computational Bounds

Fabrizio Luccio, Linda Pagli

June 28, 1996

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

An Insight on PRAM Computational Bounds*

Fabrizio Luccio and Linda Pagli
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy

luccio@di.unipi.it   pagli@di.unipi.it

Abstract

We make two contributions to the theory of PRAM computation, focusing on the problem of computing the Boolean OR (to which most basic problems can be reconducted). First we critically discuss the ideal PRAM model generally adopted to prove parallel time bounds, and introduce the natural definition of *simple* PRAM. We prove that $log_2 n$ steps are needed to compute the OR of n bits in the CREW variant of this model, and that this bound is tight. This implies that some acclaimed "surprising" results in PRAM theory depend strictly on the properties of the model, and not on the computed function. As a second contribution we show how to simplify the most advanced known lower bound proof for computing the OR [3, 2]. The new proof scheme does not introduce new results, but is aimed to enhancing the comprehension of this difficult subject. We also justify a "full information" functioning, generally adopted without discussion.

Keywords. Parallel algorithms, CREW PRAM, Lower bound theory.

1 The simple PRAM

To capture the unavoidable sequential nature of a problem, PRAM lower bounds have been studied on the so called "ideal PRAM" model, where the computational power of each processor and the size of the cells of the shared memory are unbounded. In view of such a "hardware" power, the analysis is directed to the bounds on the number of data movements between processors and memory. The seminal contributions in this line of research are due to Cook, Dwork and Reischuk [1] for CREW PRAMs, and to Snir [8] for EREW PRAMs. Virtually all subsequent works are deeply influenced by these papers, or at least mention their results as basic ones (e.g., see Parberry [7], Jaja [5], Karp and Ramachandran [6]). After some variations and improvements, a fundamental advancement over the results

*This work has been supported in part by MURST of Italy.

of [1] was presented by Kutylowski [3], and then by Dietzfelbinger, Kutylowski and Reischuk [2], who definitely settled the major computational bounds.

As known from [1], lower and upper bounds for many problems like sorting and addition can be derived from the bounds for OR. Therefore, we direct our attention to the OR function. [1] starts with a very bright algorithm to compute the logical OR of n bits in $log_{2.62}n + O(1)$, about $.72 log_2 n$ steps. The authors, and all the following contributors on this subject, find their result "surprising" because it beats the obvious upper bound of $log_2 n$, due to the fact that some processors communicate information to memory "by not writing" into it. Although the feature of "not writing" is essential in the algorithm of [1], and crucially affects the proofs of all the subsequent theorems, we think that the above presentation is misleading. In fact, the performance of the algorithm, as well as the proofs of all lower bounds, depends on the adopted PRAM model, that allows two consecutive register transfers in one step. Without this provision, the $log_2 n$ bound cannot be beaten even by "not writing" (see below). This observation bears many consequences.

One step of computation of an ideal CREW PRAM consists of the three (possibly void) operations of read, compute, and write. Formally, for each processor P_i, let $Q_i(t)$ denote the internal state (register contents) when step t is completed. Let M_j denote a memory cell (in a write operation), or its contents (in a read operation), and let $M_j(t)$ denote the same entities after step t. The operations of P_i at step $t+1$ are as follows:

1. *Read.* M_j is transferred into P_i. The value of j is a function of $Q_i(t)$.
2. *Compute.* P_i computes four functions δ, σ, ω, ϕ. $\delta(t+1) = \delta(Q_i(t), M_j(t))$ gives the next state, that is $Q_i(t+1) = \delta(t+1)$. $\sigma(t+1) = \sigma(Q_i(t+1))$, $\omega(t+1) = \omega(Q_i(t+1))$, and $\phi(t+1) = \phi(Q_i(t+1))$, respectively give the value to be written in memory in the next operation 3, the address where to write, and the (binary) decision whether or not to write. Note that σ, ω and ϕ are computed as functions of the new value $Q_i(t+1)$.
3. *Write.* If $\phi(t+1) = 1$, the value $\sigma(t+1)$ is transferred into $M_{\omega(t+1)}$. If $\phi(t+1) = 0$, no transfer is done.

As noted above, in a single step we have a transfer of information from a memory register to a processor register through the function δ, and then a transfer from the same processor register to a memory register through the function σ. Let us call this a *double* (ideal CREW) PRAM. Computing the OR of n bits in less than $0.5 log_2 n$ steps (and not $log_2 n$ steps) would be surprising, which is not the case. We then put the following definition, already introduced in a preliminary paper [4]:

Definition 1 *An (ideal CREW) PRAM is* simple *if, at each step $t+1$, a processor P_i reads the contents of a memory cell $M_j(t)$; computes the next state $Q_i(t+1) = \delta(Q_i(t), M_j(t))$ and the functions $\sigma(t+1) = \sigma(Q_i(t))$, $\omega(t+1) = \omega(Q_i(t))$, $\phi(t+1) = \phi(Q_i(t))$; and writes $\sigma(t+1)$ into $M_{\omega(t+1)}$ iff $\phi(t+1) = 1$.*

Compared with the double PRAM, note that we now compute σ, ω and ϕ as functions of the present state $Q_i(t)$, instead of the next state $Q_i(t+1)$. That is, we do not have two consecutive register transfers in the same step.

The mentioned algorithm of [1] for computing the OR of n bits can be easily adapted to a simple PRAM of n processors $P_1, ..., P_n$ [4] (for our purpose, a simple EREW PRAM suffices). The input is initially allocated in cells $M_1, ..., M_n$, and the output is built in M_1. Each processor P_i contains an internal binary register Y_i that codes one of the two possible internal states of the processor ($Y_i(t)$ codes $Q_i(t)$). Let:

$H_0 = 0,\ H_t = 2^{t-1} \quad \text{for } t \geq 1.$

For $T > 1$ we have:

Algorithm SIMPLE-OR

Initial conditions: $Y_i(0) = 0$ for $1 \leq i \leq n$;

for $1 \leq i \leq n$ **do** in parallel

  for $t := 1$ **to** T **do** in sequence

    $Y_i(t) \leftarrow Y_i(t-1) \vee M_{i+H_{t-1}}(t-1)$,

    if $Y_i(t-1) = 1$ **then** $Y_i(t-1) \rightarrow M_{i-H_{t-1}}(t)$

    where the operations are skipped for all M_j with $j < 1$ or $j > n$.

We have:

Theorem 1 *The OR of n bits can be computed with a simple PRAM of n processors in $\lceil log_2 n + 1 \rceil$ steps.*

Proof. Algorithm SIMPLE-OR converges in at most $\bar{T} = \lceil log_2 n + 1 \rceil$ steps. Consider the (worst) case of an input with exactly one 1 in M_n. It is easy to show that, after step t, exactly H_t processors $P_{n-H_t-1}, ..., P_n$, and H_t memory cells $M_{n-H_t-1}, ..., M_n$, have been reached by the 1 in M_n. To propagate the 1 into M_1 we then need $\bar{T} = \lceil log_2 n + 1 \rceil$ steps. We must now prove that the same bound holds for any input. First assume that the input has a unique 1 in M_i, $i \neq n$. The algorithm propagates this 1 along the cells $M_{i-1}, ..., M_1$, affecting H_t cells at step t. At step $T' = \lceil log_2 i + 1 \rceil \leq \bar{T}$, M_1 contains a 1, that is not altered in the next steps $T' + 1$ up to $\bar{T}$. If there are several 1's in the input, the leftmost 1 in position i reaches M_1 at time T', while the next steps do not alter this 1, due to the "**if** $Y_i = 1$ **then write**" statement. □

Note that in algorithm SIMPLE-OR some processors communicate information "by not writing" in memory, as it happened in the algorithm of [1]. Yet, we will prove that the present bound $log_2 n$ cannot be beaten.

2 The "heap of subcubes"

The computation of the OR function terminates when cell M_1 "knows" all the n input bits [1]. This is equivalent to say that M_1 must know the values of n Boolean variables contained in $M_1, ..., M_n$, hence the problem is studied in $B^n = \{0,1\}^n$. At the beginning the information contained in each cell M_i is the value of one such variable, which allows merely to infer that the problem instance is a point of a proper subcube B^{n-1}. This set is refined in the subsequent steps by the exchange of information between processors and memory, until it reduces, for M_1, to a single point B^0. In this section we present a family of sets that will be used to describe this process, leading to the lower bound proofs of section 3. This approach is essentially due to Kutylowski [3], however, we believe that our treatment is much simpler and more intuitive. (Indeed, in [3] Kutylowski himself stated that his treatment "has no immediate intuitive motivation". The technique of [2] is different and more complex).

Definition 2 *A* heap of subcubes *(shortly* heap*) of order m, $0 \leq m \leq n$, denoted by S^m, is a subset of B^n built as:*
1. a subcube B^m, $m \leq n$; or
2. the union of some disjoint heaps S_i^m of order m, denoted by $\bigcup_D S_i^m$; or
3. the complement $\neg S^m$ of a heap of order m.

A heap S^m is built from a set of subcubes $B_1^m, ..., B_r^m$, $r \geq 1$, through a *derivation tree* T of the form shown in figure 1 (each internal node of T corresponds to the application of one of the points 1, 2 or 3 of definition 2). The following properties of heaps are important for our purposes.

Lemma 1 *$|S^m| = u \cdot 2^m$, $u = 0, 1, ..., 2^{n-m}$.*
Proof. The derivation rule of definition 2 point 1 generates subsets of 2^m points. The derivation rules of points 2 and 3 generate subsets of $v \cdot 2^m$ points, $v = 0, 1, ..., 2^{n-m}$ ($v = 0$ occurs by complementation of $S^m = B^n$, where B^n is obtained as the union of 2^{n-m} disjoint heaps of order m). □

Lemma 2 *Given S^m, with $m > 1$, and any $k < m$ there is an S^k such that $S^m = S^k$ (we have $|S^k| = u \cdot 2^{m-k} \cdot 2^k$).*
Proof. If $S^m = B^m$, we have $B^m = \bigcup_D B_i^k = \bigcup_D S_i^k = S^k$, for any $k < m$. If $S^m \neq B^m$, consider a derivation tree for S^m (figure 1), and apply the above

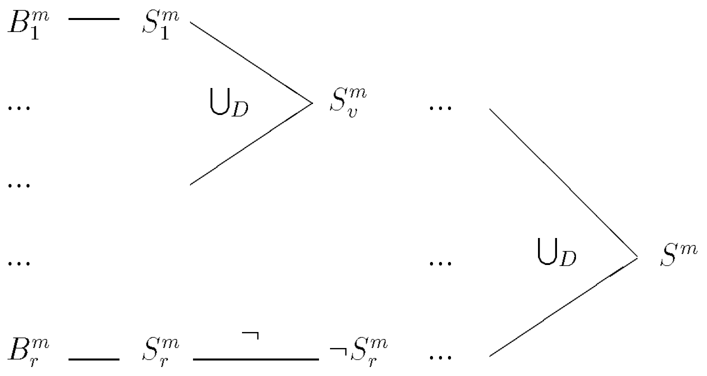

Figure 1: Derivation tree T for a heap S^m

reasoning to the generation of the lowest layer of internal nodes from the leaves, to obtain S_i^k from B_i^m, $1 \le i \le r$. Then continue the derivation up the tree applying rules 2 and 3 of definition 1. Since these rules preserve the value of k, we obtain S^k at the root. $\square$

We can now state:

Theorem 2 *For any pair of heaps S^h, S^k with $h + k \ge n$ we have: $S^h \cap S^k = S^{h+k-n}$.*

Proof. Consider two derivation trees T_h, T_k for S^h, S^k, and the numbers N_h, N_k of internal nodes of such trees. The proof is by induction on $N = N_h + N_k$.
Basis. $N_h = N_k = 1, N = 2$. We have $S^h = B^h$, $S^k = B^k$, hence $S^h \cap S^k = B^{h+k-n}$ by a known property of subcubes.
Inductive step. For $N > 2$, w.l.o.g. let $N_k > 1$, and assume that the theorem holds for any value up to $N - 1$. The root S^k of T_k may be generated as the disjoint union of some heaps S_i^k, $1 \le i \le r$ (definition 2 point 2), or as the complement of a given heap S_1^k (definition 2 point 3). We have:
Case 1. $S^h \cap S^k = S^h \cap (\bigcup_D S_i^k) = \bigcup_D (S^h \cap S_i^k)$. Noting that each derivation tree for S_i^k has a number of internal nodes $< N_k$, we have by induction: $\bigcup_D (S^h \cap S_i^k) = \bigcup_D S_i^{h+k-n} = S^{h+k-n}$.
Case 2. $S^h \cap S^k = S^h \cap (\neg S_1^k) =$ (by elementary properties of sets) $\neg \bigcup_D (\neg S^h, S^h \cap S_1^k) =$ (by definition 2 point 3, and lemma 2) $\neg \bigcup_D (S_2^{h+k-n}, S^h \cap S_1^k)$. Noting that the derivation tree for S_1^k has $N_k - 1$ internal nodes, we have by induction: $\neg \bigcup_D (S_2^{h+k-n}, S^h \cap S_1^k) = \neg \bigcup_D (S_2^{h+k-n}, S_3^{h+k-n}) =$ (by definition 2 points 2,3) S^{h+k-n}. $\square$

3 The lower bound proof

As already mentioned, any algorithm for computing the OR proceeds through successive refinements of subsets of B^n, which are known to the processors and to the memory cells. The lower bound analysis is aimed to prove that a certain number of consecutive transfers of information between processors and memory is needed to refine the subset known to cell M_1 from B^{n-1} to B^0. In fact, until this condition is not satisfied, M_1 cannot have full knowledge of the problem instance, hence the OR cannot be computed on the input 00...0.

First we assume a "full information" functioning as in [3, 2], where processors and memory cells exchange all the information they have collected so far, and build the most refined subsets at each step. The meaning and validity of this assumption will be discussed in detail after the proof of the main theorem 3. In particular, if, at a given step, processor P_i knows subset X, and cell M_j knows subset Y, we have:

Rule 1: if P_i reads from M_j, the subset known to P_i is refined to $X \cap Y$;

Rule 2: if P_i writes into M_j, the subset known to M_j is refined to $Y \cap X$ (in the model of [1] Y is overwritten by X, see the discussion below);

Rule 3: if no processor writes in M_j, let $P_1, ..., P_r$ be the processors that may write into M_j at that step, and $X_1, ..., X_r$ be the subsets known to such processors when they write in M_j; by the Exclusive Write hypothesis $X_1, ..., X_r$ must be disjoint; if no processor writes, the subset known to M_j is refined to $Y \cap (\neg \bigcup_D (X_1, ..., X_r))$.

As anticipated, the following theorem gives a simpler proof of a known lower bound for the standard CREW PRAM (point i). It also gives the companion proof for the simple CREW PRAM (point ii).

Theorem 3 *To compute the OR of n bits:*
i. an (ideal double) CREW PRAM requires at least $log_{2.62}n + O(1)$ steps;
ii. a simple CREW PRAM requires at least $log_2 n + 1$ steps.

Proof. At the beginning n memory cells know one subset B^{n-1} (i.e., one input variable), and all processors know B^n (i.e., they have no information on the input). Since the information is exchanged with rules 1, 2, 3, by definition 2 and theorem 2 we have that, at any step, the subsets known by the processors and by the memory cells are heaps, no matter which algorithm is used. Let $S^{K(t)}$ and $S^{L(t)}$ denote the smallest heaps respectively known by a memory cell and a processor after step t. For the ideal double PRAM we have:

$L(0) = n$, $K(0) = n - 1$,

$L(t) = L(t-1) + K(t-1) - n$, for $t \geq 1$ (rule 1, theorem 2),

$K(t) = K(t-1) + L(t) - n$, for $t \geq 1$ (rule 2 or 3, theorem 2),

whose solution for $K(t)$ is readily computed as $K(t) = n - F_{2t+1}$, where F_{2t+1} is the Fibonacci number of index $2t + 1$. For the simple PRAM we have:

$L(0) = n$, $K(0) = n - 1$,

$L(t) = L(t-1) + K(t-1) - n$, for $t \geq 1$ (rule 1, theorem 2),

$K(t) = K(t-1) + L(t-1) - n$, for $t \geq 1$ (rule 2 or 3, theorem 2), whose solution for $K(t)$ is readily computed as $K(t) = n - 2^{t-1}$. We must now determine the minimum number T of steps needed to refine the subset $S^{K(t)}$ known to M_1, from B^{n-1} to B^0. By lemma 1, this may happen only for $K(T) = 0$. Combining this condition with the solutions of the above recurrences we immediately find the bounds stated in the theorem. □

The full information functioning has been taken in the literature without further discussion, under the implicit assumption that communicating all knowledge at each step does not affect the lower bound proof. What is not at all obvious, however, is that building the most refined subsets at each step (in particular, taking the intersection of subsets as in rules 1,2,3) does not increase the number of such steps. This point seems not to have been raised before. If on the one hand the computation terminates when the subset in M_1 reduces to a single point, on the other hand the logical complementation required by a "not writing" operation (rule 3) implies that, if a processor P_i writes into M_j a subset X_i' instead of the most refined subset X_i (i.e., $|X_i'| > |X_i|$), then the subset $Y \cap (\neg > \bigcup_D(X_1, .., X_i', .., X_r))$ built in M_j when no processor writes is smaller than the one built using X_i. This seems to jeopardize the lower bound proof. We now prove that all the possible subsets built in a PRAM computation are heaps of a certain class, and then we legitimate the assumption of full information functioning.

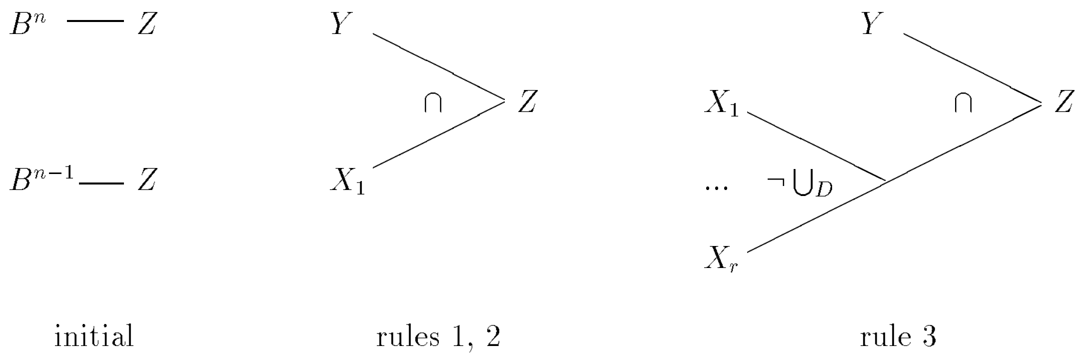

Figure 2: Computation tree U for a subset Z built in a CREW PRAM

At any step the subset Z known to a processor or to a memory cell is built through a *computation tree* U of the form shown in figure 2, where Y and $X_1, ..., X_r$ are subsets inductively built in the same way. For the full information assumption, U is known along with Z. A less refined subset Z' can be built from U by substituting B^n for B^{n-1} in a leaf (i.e., ignoring an input variable at that point); and/or substituting B^n for Y in an internal node (i.e., letting $Y \cap X = X$, as for example in an overwriting memory operation); and/or excluding some of the X_i from the disjoint union of rule 3. The subset Z' is then propagated, taking the place of Y or X_i in the computational tree of another subset. According to definition 2, lemma 2 and theorem 2, each subset thus built is *a heap of order not less than the one of the corresponding most refined subset.* This legitimates the assumption of full information functioning, since a heap of order 0 must be built in M_1 to terminate any algorithm.

Note that Z' can be used for any purpose in the PRAM program, including in the condition of a "write/not-write" **if** statement. The reader may consider the following example. For Boolean variables $x_1, ..., x_n$, let $n-3$ be the order of the most refined heaps known to the processors at a given step, and let Z be one such a heap. In particular, let $Z = B^{n-3}$ be the subcube corresponding to the actual values of x_1, x_2, x_3 in the problem instance. When executing the following statement, a processor P_i knows Z, but uses also the less refined heaps $Z' = B^{n-1}$ corresponding to the particular value $x_1 = 1$, and $Z'' = B^{n-2}$ corresponding to the actual values of x_2, x_3:

if $x_1 = 1$ (i.e., if the input point is in Z')
    then write the values of x_2, x_3 (i.e., write Z'') in M_j
    else write the values of x_1, x_2, x_3 (i.e., write Z) in M_k.

For a problem instance with $x_1 = 1$, P_i writes Z'' into M_j. Since $|Z''| > |Z|$, and the order of Z'' is greater than the one of Z, writing Z'' instead of Z does not endanger the lower bound proof. For a problem instance with $x_1 = 0$, the information passed from P_i to M_j is that the input point belongs to $\neg Z'$ (i.e., $x_1 = 0$). We now have that $|\neg Z'| < |\neg Z|$, however, the order of $\neg Z'$ is equal to the order of Z', hence is greater than the order of Z. Again, using Z' does not endanger the lower bound proof.

As a final remark note that our approach can be extended to determine a lower bound for computing any Boolean function, as already done in [3, 2]. First note that any subset $X \subseteq B^n$ is a heap (as a limit, X is a heap of order 0 built as the union of single points). For any function f we then have $f = 1$ in a heap X_f of proper order (i.e. maximum order: recall lemma 2), hence the computation cannot terminate until the heap known to M_1 is refined to that order. A lower bound can then be computed for f, for any PRAM model, with an immediate extension of the proof of theorem 3, although a nontrivial task maybe the one of determining the order of X_f. In particular, for the functions with X_f of order 0 we obtain the same lower bound of the OR function.

References

[1] S. Cook, C. Dwork and R. Reishuck. Upper and Lower Bounds for Parallel Random Access Machines without Simultaneous Writes. *SIAM J. Comp.*, 15 (1986) 87-97.

[2] M. Dietzfelbinger, M. Kutylowski and R. Reischuck. Exact Lower Time Bounds for Computing Boolean Functions on CREW PRAMs. *J. Comp. Sys. Sci.* 48 (1994) 231-254.

[3] M. Kutylowski. Time Complexity of Boolean Functions on CREW PRAMs. *SIAM J Comp.* 20 (1991) 824-833.

[4] F. Luccio and L. Pagli. Computational Bounds for the Simple and the MRMW PRAM. *Proc. IEEE Symp. Par. Distr. Proc.* (1994) 552-556.

[5] J. Jaja. *An introduction to Parallel Algorithms*, Addison Wesley, Reading, MA, 1992.

[6] R. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, Ed., *Handbook of Theoretical Comp. Science*, North Holland, Amsterdam (1989) 869-941.

[7] I. Parberry *Parallel Complexity Theory*, Pitman, London, 1987.

[8] M. Snir. On parallel searching. *SIAM J. Comp.* 14 (1985) 688-708.